LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TH-229287

Determining Application Runtimes Using Queueing Network Modeling

M.L. Elliott

March 22, 2007

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Meeting the Deadline

Determining Application Runtimes
Using Queueing Network Modeling

A thesis presented to the
Faculty of the Computer Science Department,
University of San Francisco,
in partial fulfillment of the requirements for the
Degree of Master of Science in Computer Science

By
Michael L. Elliott
December 14, 2006

UCRL-TH-229287

This research supported in part by UC/LLNL subcontract #B546340.
Portions of this work were performed under the auspices of the U.S. Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Thesis Approval

This thesis, written by:

Michael L. Elliott

B.S. Electronics Engineering Technology, DeVry University, Kansas City, MO, 2002
B.A. Philosophy, University of Missouri, Columbia, 1997

under the guidance of the Thesis Advisory Committee, and approved by all its members, has been presented to and accepted by the Dean of the College of Arts and Sciences, in partial fulfillment of the requirements for the degree of:

Master of Science
In
Computer Science

_____

Jennifer Turpin, Ph.D. Date
Dean
College of Arts and Sciences, USF

Thesis Committee:

_____

Jeff T. Buckwalter, Ph.D. Date
Associate Professor and Committee Chair
Department of Computer Science, USF

_____

David J. Galles, Ph.D. Date
Associate Professor
Department of Computer Science, USF

_____

Dr. rer. nat. Martin Schulz Date
Computer Scientist
CASC, Lawrence Livermore Nat'l Lab.

Vita Auctoris

Name:	Michael L. Elliott
Date of Birth:	August 26, 1972
High School:	Jefferson City Senior High School, Jefferson City, MO
Graduated:	June, 1990
Baccalaureate School:	University of Missouri – Columbia
Degree Awarded:	B.A. in Philosophy, December, 1997
Baccalaureate School:	DeVry University – Kansas City, Missouri
Degree Awarded:	B.S. in Electronics Engineering Tech., February, 2002

Acknowledgements and Dedications

My thanks to the faculty and staff of the University of San Francisco and Lawrence Livermore National Laboratory, without whom this research could not have been conducted.

I would also like to thank the other students involved in this project for picking up the slack when I had to focus my time elsewhere.

Special thanks to the members of my thesis committee, who have read countless revisions of this thesis, and provided invaluable feedback and support.

Very special thanks and my eternal gratitude to my committee chair and mentor, Dr. Jeff Buckwalter, whose belief in me allowed me to excel to new heights.

Finally, I dedicate this thesis to my grandmother, parents, and partner. Without their support, sacrifices, and faith in me, I would not be where I am today.

Abstract

Determination of application times-to-solution for large-scale clustered computers continues to be a difficult problem in high-end computing, which will only become more challenging as multi-core consumer machines become more prevalent in the market. Both researchers and consumers of these multi-core systems desire reasonable estimates of how long their programs will take to run (time-to-solution, or TTS), and how many resources will be consumed in the execution. Currently there are few methods of determining these values, and those that do exist are either overly simplistic in their assumptions or require great amounts of effort to parameterize and understand. One previously untried method is queueing network modeling (QNM), which is easy to parameterize and solve, and produces results that typically fall within 10 to 30% of the actual TTS for our test cases. Using characteristics of the computer network (bandwidth, latency) and communication patterns (number of messages, message length, time spent in communication), the QNM model of the NAS-PB CG application was applied to MCR and ALC, supercomputers at LLNL, and the Keck Cluster at USF, with average errors of 2.41%, 3.61%, and -10.73%, respectively, compared to the actual TTS observed. While additional work is necessary to improve the predictive capabilities of QNM, current results show that QNM has a great deal of promise for determining application TTS for multi-processor computer systems.

Preface

Conventional commodity serial processors are nearing their limits for speed gains and increased computational ability, as evidenced by Intel's decision to discontinue research and development of higher clock-rate serial processors in favor of lower rate multi-core architectures[1]. Power consumption and heat dissipation make further advances by increasing clock speeds and transistor density unlikely except for very specialized applications. Already, major commercial processor manufacturers are exploring the use of multiple-core technologies to feed consumers' desires for faster processing times and more feature rich (and therefore computationally expensive) environments. This effort, led by the scientific community and driven by the market for processing in the gaming community, has sponsored a new paradigm in software engineering. As multiple processor machines become more and more common, programmers are moving away from the serial approach, with programs considered as a linear sequence of operations, to a parallelized approach, with programs written to run across many processors or cores, each solving a small subproblem related to the original problem. However, up to now, it has been difficult to accurately model the time-to-solution (TTS) for such parallel systems. This information is highly desirable for a number of reasons, including: users wish to know how long their programs will run, efficient batch scheduling, and resource utilization efficiency. Recent research at the University of San Francisco (USF) in Queueing Network Modeling (QNM) suggests that QNM is a useful methodology for this problem of estimating TTS.

[1] [Kanellos, 2004]

Table of Contents

Table of Figures

Table of Equations

Table of Tables

I – Introduction

In today's world, the need for computing power is becoming more pressing daily. Our need to process, analyze, and store data is quickly exceeding the capabilities of small self-contained serial machines, such as the modern desktop PC. Initially, the creation of supercomputers filled this gap: large-scale self-contained parallel machines. However, current markets, as well as the costs to develop and maintain such machines, are quickly making such machines less common, used only in highly specialized environments. A third type of machine exists, however. This relatively new type of machine, known as a cluster and built from common, and often inexpensive, commodity components for computation, and either commodity or specialized interconnects, is easy to construct, inexpensive compared to specialized, custom machines, and is incredibly pervasive in the market. However, how well do clustered machines work?

There have been many attempts to quantify the performance of clustered computers. One approach, Queueing Network Modeling (QNM), is a little tried, but potentially useful means of modeling such systems. QNM, which has its beginnings in the modeling of traffic patterns, has expanded. It is now useful for modeling everything from CPU and disk services, to computer systems, to service rates in store checkout lines. This history of successful usage, as well as the correspondence of QNM components to commodity clusters, suggests that QNM is a useful tool for both the cluster designer, interested in the best price/performance ratio, and the user of existing machines, interested in performance rates and time-to-solution.

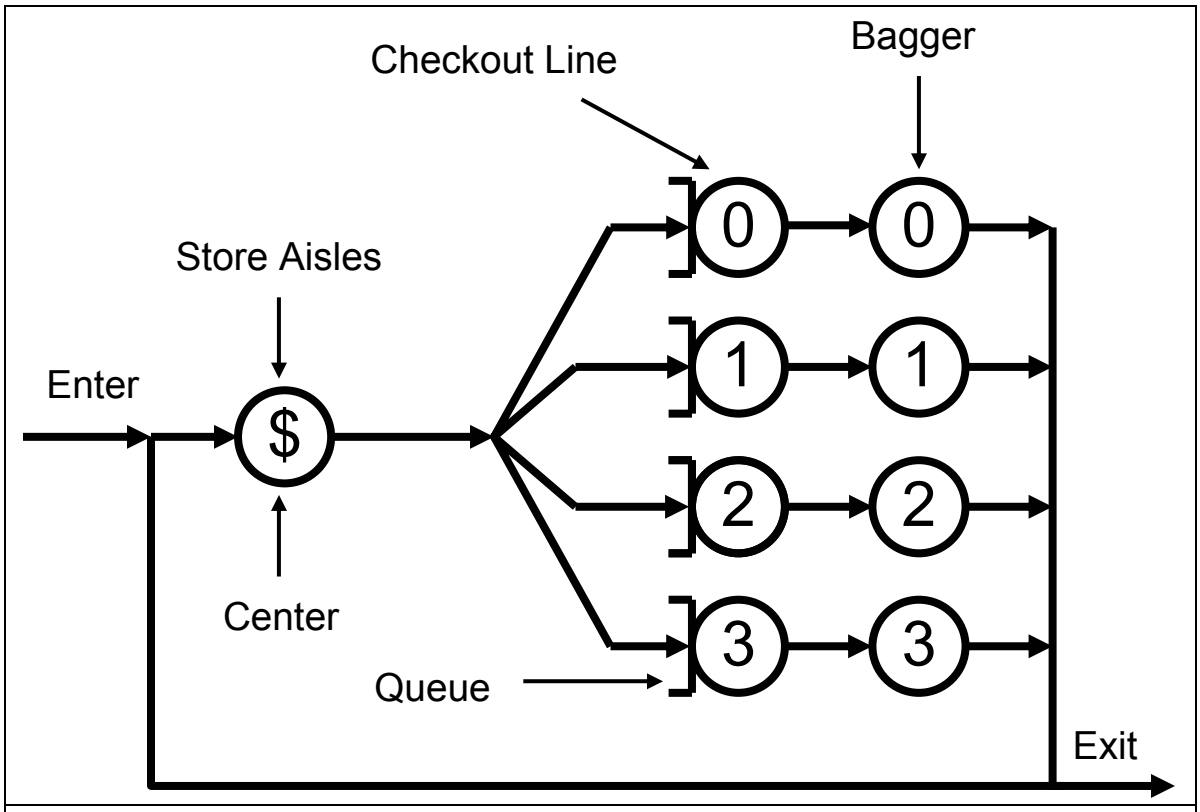

Figure 1 – Simple QNM Parameterization

Queueing Network Modeling is an approach to complex system modeling where a network of analytically evaluated queues represents the computer.[2] Figure 1 above shows a simple QNM model. In this model, customers are shopping in a store. Customers enter the store and proceed to wander through the aisles, modeled as a delay center, assuming each customer takes on average the same amount of time to shop. When a customer finishes shopping, they proceed to the checkout line, where they may have to queue behind other customers. The time it takes a customer to go from the end of the queue to the head of the queue is dependent on the number of customers preceding. Therefore, the checkout line is a queue, and the cashier is a queueing center. Assuming,

[2] [Lazowska, 1984]

again on average, the bagger requires the same amount of time to service each customer, the bagger is a delay center. We assume a large pool of store clerks who can do bagging, so that customers do not need to wait for bagging. After visiting the bagger, the customer either exits the store, or returns for more shopping. If customers are free to enter the store at any time, and there is no limit to the number of customers in the store at any given time, this example represents an open QNM model. If, on the other hand, there is a limit to the number of customers that may shop at any given time (i.e.: a maximum occupancy), and new customers may not enter the store once this limit is reached unless a shopper exits, then this example represents a closed QNM model. Chapter II, Section C, Subsection 7 explores this relationship as applied to parallel computers further.

The general goal of this research is to explore the hypothesis that QNM is an appropriate approach for estimating the runtimes of applications in parallel computers. This thesis will focus on analyzing the viability of QNM as a model for actual machine performance. We will collect and present data on actual machine behavior, and then we will run the QNM models, and compare the results with the measured machine performance to see how accurately QNM can model the observed behavior.

The remainder of this thesis is organized into the following sections:

II. Background and Related Research

Summary of work related to this research, both historical and recent.

III. Methodology and Experimental Environment

Description of the various components and methods used to create, test, and validate the QNM model.

IV. Results and Analysis

An analysis of collected data and report of the findings.

V. Problems Encountered and Further Research

Description of problems encountered during the research process, and areas for further research on the QNM model.

VI. Summary and Conclusion

Summary of the research results and conclusions drawn from result analysis.

II – Background and Related Research

A – Scalability Review

In parallel computing, there are two interconnected facets of scalability, both measured by time to solution. The first is the Strong Scaling Problem, sometimes called Problem-Constrained Scaling. In this type of scaling, a given application with a given input set runs on increasing numbers of nodes. Generally, one would expect that a system with more nodes should produce smaller times-to-solution than a system with fewer nodes. However, most applications on large systems begin to experience larger and larger overheads, due in part to the larger number of inter-node messages flowing over the network and also, in part, to load imbalance. Upon reaching some critical number of nodes, not only does the addition of new nodes fail to increase performance, additional nodes may actually decrease performance. Only embarrassingly parallel networks, such as seti@home, boinc, and others have largely overcome this limitation. For these applications, the computations performed by individual nodes have little or no relationship to computations on other nodes, in effect making them embarrassingly parallel. On the other hand, for tightly coupled applications that require interaction and communication, this limitation is very real, and places an effective cap on the size and type of computations these clustered systems may perform.

The second facet of parallel computing is the scalability of the computational algorithm as the problem size or input size increases. Time to solution for problems submitted to a cluster will typically increase as the problem or input set increases in size. Adding additional nodes to the computation can mollify this effect, traditionally known as the

Weak Scaling Problem. It involves balancing the increase in solution time from the expanded input with and the reduction in solution time from the addition of extra computational nodes. This version of the problem is Classic Weak Scaling or Weak Scaling II in this thesis, and is sometimes called Time-Constrained Scaling. A subversion of this problem, referred to as Weak Scaling I in this thesis, deals with the increase in time to solution as the input size increases but number of nodes remains constant.

B – Programming Models Review

Every user knows that computers run programs, and this ability gives computers their power. This begs the question: what is a program? A program is a set of instructions given to the computer that allow it to receive information from the outside world, manipulate that information in some meaningful manner, and use the manipulated information to take action in the outside world. (For our purposes, the outside world is anything that exists outside the processing unit, including disk drives, printers, keyboards, main memory, etc.) Several different methods of providing and organizing these instructions exist, and are explored in the following subsections.

1 – Electro-mechanical (Hard) Programming

The first computers were programmed by mechanically establishing electrical connections (hard connections, thus hard programming) between various components. To change the program required rewiring the machine to reflect the new connections. A simple example of this is a light switch. The switch accepts input (flipping the switch), manipulates the input (creating an electrical connection within the switch), and takes

some action (electricity flows, and the light comes on/goes off). To get the switch to perform in another manner (say three positions instead of two) requires opening the switch and modifying the internal working of the switch. (This is not recommended, as it can be EXTREAMLY DANGEROUS!) This method is still used in chip design, industrial applications, and many common items. The read/decode/execute/store routine in a computer's CPU is such a program physically embedded into the chip. Electro-mechanical programming is generally much less expensive than textual programming, discussed following, which relies on electro-mechanically programmed devices (software does nothing if not run on the proper hardware). These devices, however, are often difficult to modify, and wear out through regular use, eventually leading to failure. Programmable gate arrays, while being easily modified, are a form of electro-mechanical programming, as it requires physically changing the interior configuration of the array to change the program. Programmers often design electro-mechanical programs graphically using specialized CAD software. Parallel processing requires the addition of new hardware to handle the parallel input.

2 – Textual (Soft) Programming

As computer science began to evolve, the stored-program concept emerged, giving rise to textual programming. In this type of programming, actions themselves are not hardwired into the machine, so much as the potential for actions. Instructions are read, temporary (soft) connections are made, and various circuits are activated or deactivated, on the fly. This is what a typical person thinks of when mentioning the word "program." Textual programming results in much greater flexibility, both in modifying the actions taken by the machine and in determining which program will execute. Programs of this

type are tightly bound to the type of device they will properly run on, however. Textual programming is often done using word-processing software (the text), though graphical means do exist.

Because soft programming is so dependent on its hardware, several different programming paradigms exist to create soft programs. The major paradigms are explored below.

2.1 – Serial Programming

Serial programming looks at instructions the way one would the directions in a cookbook: as a set of logical steps done strictly in order, until no more steps remain. Until recently, this was, by far, the dominant paradigm. Most serial computers, at least from the programmer's and user's views, are single-instruction, single-data (SISD). One instruction executes at a time, and it executes on a single piece of information. (In reality, modern computers are able to optimize code on-the-fly using pipelines and code reordering, but this behind-the scenes work is unseen by the user or programmer.) Nevertheless, high-end specialty computers of this type exist, known as vector computers. These computers are single-instruction, multiple-data (SIMD). They still execute one instruction at a time, but it affects multiple pieces of information.

2.2 – Parallel Programming

Parallel programming is less rigid in the ordering of instructions than serial programming. Instead, parallel programs are the "efficiency experts" of programming. They see instructions based on the data dependencies and conflicts. Those areas where data in one portion of the program is dependent on instructions executed elsewhere in the

program run (more or less) serially, and those areas where there are no dependencies and there are no data conflicts run concurrently (in parallel). Depending on the underlying hardware, this concurrency is achieved in one of two basic means: pseudo-parallelism and true parallelism.

2.2.1 – Pseudo-parallelism

Programmers achieve pseudo-parallelism when a parallel programming paradigm is used to program a SISD or SIMD machine. Since the machine can only execute one instruction at a time, true instruction parallelism is impossible. However, using a scheduler program it is possible to swap multiple programs onto the processor in a very brief period. If done quickly enough, it will appear to the user as though the programs are running concurrently. Examples of pseudo-parallelism are threads and multi-processing common in most personal computers.

2.2.2 – True Parallelism

True parallelism requires that multiple processors be available, and that each processor be capable of executing a different instruction on different data (multiple-instruction, multiple-data, or MIMD[3]). Programmers have many different means of exploiting this property, the major means of which we detail below.

2.2.2.1 – Message Passing

The Message Passing Interface (MPI) is currently the most common form of programmed parallelism. MPI is an interface, with no specific implementation

[3] While multiple-instruction, single-data (MISD) machines are possible in theory, few, if any, practical applications for such a machine exist.

requirements, other than the interface is maintained. Thus, there are several different implementations of MPI, both public and proprietary. MPICH, LAM, LA-MPI, and USF-MPI are some of the more common implementations of MPI. MPI assumes that the entire program is de facto parallel, and it is the responsibility of the programmer to determine which portion of the code and data is relevant on each machine, based on a rank number given to each node on program startup. As its name implies, MPI utilizes message passing to communicate between the nodes, and contains routines to send/receive messages, synchronize the machine, establish communication patterns, and perform other cluster management tasks. MPI requires that the programmer explicitly determine the division of the data and the message passing structure. MPI was the parallel programming model used for this research.

In MPI, each node is a stand-alone entity, possessing its own memory, operating system, background processes, and other system resources. Each node receives identical copies of the parallel code to execute, and a subset of the overall data set to perform computation on. In general, the processes exchange data using messages to work on the global problem. In the NAS-PB CG code, which formed the basis of our testing, as each node finishes computation on its subset of the data, it exchanges information about the results with other nodes in the form of messages passed between the nodes. Once this communication is complete, each node then begins to reprocess the data, until the program finishes execution. At this point, the subsets of the data are recombined into the overall set, and the results are returned to the user.

2.2.2.2 – Remote Procedure Calling

In Remote Procedure Calling (RPC), each processor works independently on portions of the data. When the initiating node needs to pass data or requires a service from another node, the initiating node performs a function call to the remote node. The remote node then collects the data or performs the service and sends a procedure call return to the initiating machine. This makes RPC similar to object-oriented threading on SISD and SIMD machines.

2.2.2.3 – Shared Memory

Most means of parallelism assume that each computation node in the system is operating with its own, local, private memory hierarchy. This requires that nodes pass messages to each other when information not present locally is required, as explained in the following sections. However, there are machines in which every processor has access to a global memory hierarchy. In these shared memory machines, communication between processes occurs by writing to this global memory space, eliminating the need for message passing. Most threading implementations and multi-core commodity machines use shared memory for inter-process communication.

OpenMP, a type of shared memory parallel programming, allows the programmer to designate portions of a program as either serial or parallel, allowing the compiler to handle the details of actual parallelization. During execution, when a parallel section of code is encountered, the OMP libraries send copies of the parallel code from the head node to the remote nodes. The remote nodes then begin processing the data, using shared memory to communicate with the other remote nodes and the head node. At the end of the parallel portion of the program, the remote nodes return the data to the head node and

serial execution resumes, until the next parallel portion of the program, where the process repeats. In OMP, the programmer focuses on the areas of parallelism, and leaves the details of data division and message passing to the OpenMP libraries.

C – Benchmarking Review

Benchmarking is a means of attempting to measure a computer's performance in relation to that of other computers over a known workload, usually by some meaningful output metric such as time to solution or number of computations performed. There is much research to confirm the notion that there is not and can never be a perfect benchmark, as all users' needs are different, and no artificial means of measurement can consider all possibilities. However, benchmarks can provide useful information in one of two ways.

Some benchmarks attempt to simulate the average user by attempting typical sets of tasks undertaken in a typical computing environment, including opening text editors, calling compilers, and running complex math packages. Benchmarks of this type are very difficult to create for an "average user," as the needs of every user of a system tend to vary widely. Often, when this type of benchmark is required, one is specifically created to model a known system usage, and the results cannot be abstracted for other types of usage. New usage paradigms require a completely new benchmark.

Other benchmarks, focus on one type of machine usage, and test many different ways of carrying out that specific task and the possible ways it may be used. These benchmarks are easy to acquire, compared to the more general kind, and there has been much research into this type of benchmarking scheme.

This section will focus on pre-coded benchmarks. Another set of benchmark types, pencil and paper benchmarks, where the researcher is free to develop his or her own solution, are not considered here.

1 – Whetstone

Designed in the 1960's by Brian Wichmann in Whetstone, England, and first implemented by Harold Curnow in 1972, Whetstone is the first major synthetic benchmark. Although it includes some integer code, Whetstone functions primarily on floating point operations with particular focus on the transcendental functions such as: sin, cos, atan, log, and exp. Both scalar and vector solutions are calculated.[4]

Whetstone is specifically useful to those whose work requires many floating-point calculations on tightly bound spatially local variables, as these are the conditions where Whetstone excels.[5] Unfortunately, for users of mostly integer-based calculations, such as text editing, O/S operations, compiling, graph algorithms, and en/decrypting data, Whetstone will not offer very meaningful results.

2 – Dhrystone

R. P. Decker developed Dhrystone in 1984. As its name implies, Dhrystone is a response to the inadequacies of Whetstone. Dhrystone is CPU bound, and performs no I/O or system calls.[6]

Dhrystone is "designed to test performance factors important in non-numeric systems programming."[7] It does no floating-point operations. It is also very dependent on the

[4] [Balsa, 1997]
[5] [Bramer, 2004]
[6] [Weboped, 2004]

cache size, and systems with smaller caches will notice significant performance degradation. Dhrystone is also weak in the way it handles strings, and this may lead to unreliable results.[8]

3 – Linpack

Linpack is "a measure of a computer's floating-point rate of execution . . . determined by running a computer program that solves a dense system of linear equations."[9] Jack Dongara designed Linpack in the 1970's and it was used extensively in the 1980's (and continues to enjoy widespread use as the benchmark used to determine the Top 500 supercomputers) as a means of gauging computer performance. Originally implemented in FORTRAN, Linpack works by solving linear equations and least squares problems. Researchers are free to develop their own programs to implement the Linpack benchmarks, as long as they solve the problems defined by the Linpack specifications. The problems include linear systems with general, banded, symmetric indefinite and positive definite, triangular, and tridiagonal square matrices, as well as QR and singular value decompositions of rectangular matrices as applied to least-squares problems.[10]

Linpack makes a convenient tool for performance measurement, and is used by the Top 500 Supercomputer List, because "[b]y measuring the actual performance for different problem sizes n, a user can get not only the maximal achieved performance Rmax for the problem size Nmax but also the problem size N1/2 where half of the performance Rmax is achieved."[11] However, Linpack is not a panacea for benchmarking. Its use of memory

---

[7] [Bramer, 2004]
[8] [Bramer, 2004]
[9] [Dongara, 2004a]
[10] [Dongara, 2004b]
[11] [Mauer, 2004]

is not very efficient, resulting in a lot of overhead from data relocation.[12] Linpack is also computation bound and does not effectively evaluate the network interconnect.

4 – ASCI Purple Benchmark Suite

The ASCI Purple benchmark suite was created to guide the procurement of the ASCI Purple machine at LLNL and contains several large-scale benchmarks, which represent the planned workload for the machine. From the ASCI Purple website: "[T]he intent of these benchmarks is to measure the execution performance and compiler capabilities.... Each of the benchmark programs represents a particular subset and/or characteristic of the expected ASCI workload, which consists of solving complex scientific problems using a variety of state-of-the-art computational techniques. It is assumed that the details of hardware and/or software environment between the benchmarking configuration… may differ. Differences between the hardware and/or software environment between the benchmarking configuration… can be compensated for by coherent[ly] scaling arguments to more relevant configurations."[13] The ASCI Purple Benchmarks are written for MPI and OpenMP.

5 – NAS Parallel Benchmarks

The Numerical Aerodynamic Simulation Parallel Benchmarks (NAS-PB) are described as: "[A] small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications."[14]

[12] [Dongara, 2004a]
[13] [Purple, 2001]
[14] [NAS-PB, 2004]

Developed at NASA's Ames Research center, the benchmarks consist of two major components: five parallel kernel benchmarks and three simulated application benchmarks,[15] [16] characterized as follows:

Kernel Benchmarks:

EP	Embarrassingly Parallel: Compute bound with virtually no inter-processor communication.
MG	Multigrid: Tests both short and long distance communication.
CG	Conjugate Gradient: Tests irregular long distance communication.
FT	Fast Fourier Transform: Rigorous long-distance communication test.
IS	Integer Sort: Tests both computation speed and communication performance.

Simulated Applications:

LU	Lower/Upper: Regular-sparse, block lower and upper triangular system solution. Limited parallelism. Very sensitive to small message communication performance. Large numbers of small (40 byte) messages.
SP	Scalar Pentadiagonal: Solves scalar pentadiagional systems resulting from full diagonalization of the approximately factored scheme. Provides good load balance and coarse-grained communication.

[15] [Bailey, 1994]
[16] [Bailey, 1995]

BT Block Tridiagonal: Solves block tridiagonal systems of 5×5 blocks. Provides good load balance and coarse-grained communication.

The pre-coded MPI-based benchmarks are configurable, at compile time, for class sizes and numbers of nodes. (Pencil and paper as well as OpenMP algorithms also exist.) Class groups (S, W, A, B, C, and D) provide increasingly larger problems used to test MPI. Certain tests have restrictions on the number of processors. Processor allocations for BT and SP must be a square value (e.g. 1, 4, 9, 16 ... processors). Allocations for CG, FT, IS, LU and MG must be a power of two (e.g.1, 2, 4, 8 ... processors). There are no size restrictions for EP.

D – System Modeling and Prediction Review

Benchmarking has little value in itself, other than for comparing existing systems. However, when used with performance models, benchmark results can provide prediction of future system performance, as well as system performance under differing configurations (different problem sizes or numbers of nodes). While often simplified, as it is impossible to account for every variable in each individual system, these models frequently provide reasonable forecasts of system performance when known factors, such as number of nodes or network speed, are changed. Different modeling systems make different assumptions about the system modeled. That is, they ignore certain "inessential" factors and focus on other "important" factors. Consideration of these different models is essential for constructing a working model. We will discuss these models in the following sections.

1 – RAM

The Random Access Machine model is a favorite model for sequential computers. It consists of an unbounded number of memory cells and each cell consists of an integer of unbounded size. It includes most basic machine instructions, and assumes a constant time per instruction. When used to analyze a particular algorithm, RAM provides results in the form of time complexity measures (number of instructions executed) and space complexity measures (number of memory references made.) While RAM is not suited for parallel modeling itself, it is the progenitor of an entire class of parallel models.[17]

2 – PRAM

Parallel Random Access Machine grew from RAM. PRAM assumes an unbounded collection of RAM processors, an unbounded collection of memory cells globally shared by all processors, and an unbounded set of local registers.[18] Further, it also assumes that all processors in the machine operate synchronously and that interprocessor communication, via shared memory, requires no overhead.[19] Returned results are complexity numbers, similar to RAM.

3 – LogP

Both RAM and PRAM are machine independent. LogP handles these inadequacies by taking into account machine-specific parameters. It achieves this by utilizing measured or estimated input variables such as: the number of processors, the communication

[17] [Tvrdik, 1999]
[18] [Tvrdik, 1999]
[19] [Culler, 1993]

bandwidth, the communication delay, and the communication overhead.[20] It assumes asynchronous processors, and a maximum limit on the number of messages that may be in the system at a given time.[21] LogP effectively measures point-to-point communication. LogP, as the rest of the "Lo" family of models, derives its name from the mathematical symbols used in the model.

4 – LogGP

LogGP is an extension of the LogP model. It incorporates long messages into LogP, something not previously supported. LogGP utilizes the same inputs as in LogP, and adds an additional input, bandwidth for long messages, to the model.[22] These specifically gear LogGP toward the current trends in commodity cluster computing, where both short and long messages often occur in the network.[23]

5 – LoPC

LoPC is another logical extension of the LogP model. It does not handle long messages, as does LogGP, but it does handle contention issues with the addition of a contention parameter and the removal of the communication bandwidth parameter. LoPC works well with non-tightly synchronized systems because of its inclusion of a contention parameter, which becomes more important in less synchronized systems.[24]

[20] [Culler, 1993]
[21] [Culler, 1993]
[22] [Alexandrov, 1995]
[23] [Alexandrov, 1995]
[24] [Frank, 1997]

6 – Application Modeling

Many models are designed around modeling and predicting the behavior of particular applications, and generalizing the performance of particular machines from the collected data. The works of Allan Snavely[25], Jack Dongara[26], Ipek[27,28], and Lee[29] exemplify this type of modeling.

7 – Queueing Network Modeling

QNM, a type of application modeling, is a form of modeling in which a system reduces to a small number of relevant parameters, which can then be solved analytically using various solution techniques. Originally designed to model automobile traffic across bridges, through tunnels, and around highway interchanges, QNM underwent successful modification to apply to a wide range of systems in which queueing bottlenecks can affect performance, including computer science. Research by Lazowska, et al, shows that QNM is highly useful in modeling the performance of multiple magnetic storage devices servicing the needs of multiple computational units.[30]

[25] [Snavely, 2001]
[26] [Dongara, 2004b]
[27] [Ipek, 2005]
[28] [Ipek, 2006]
[29] [Lee, 2006]
[30] [Lazowska, 1984]

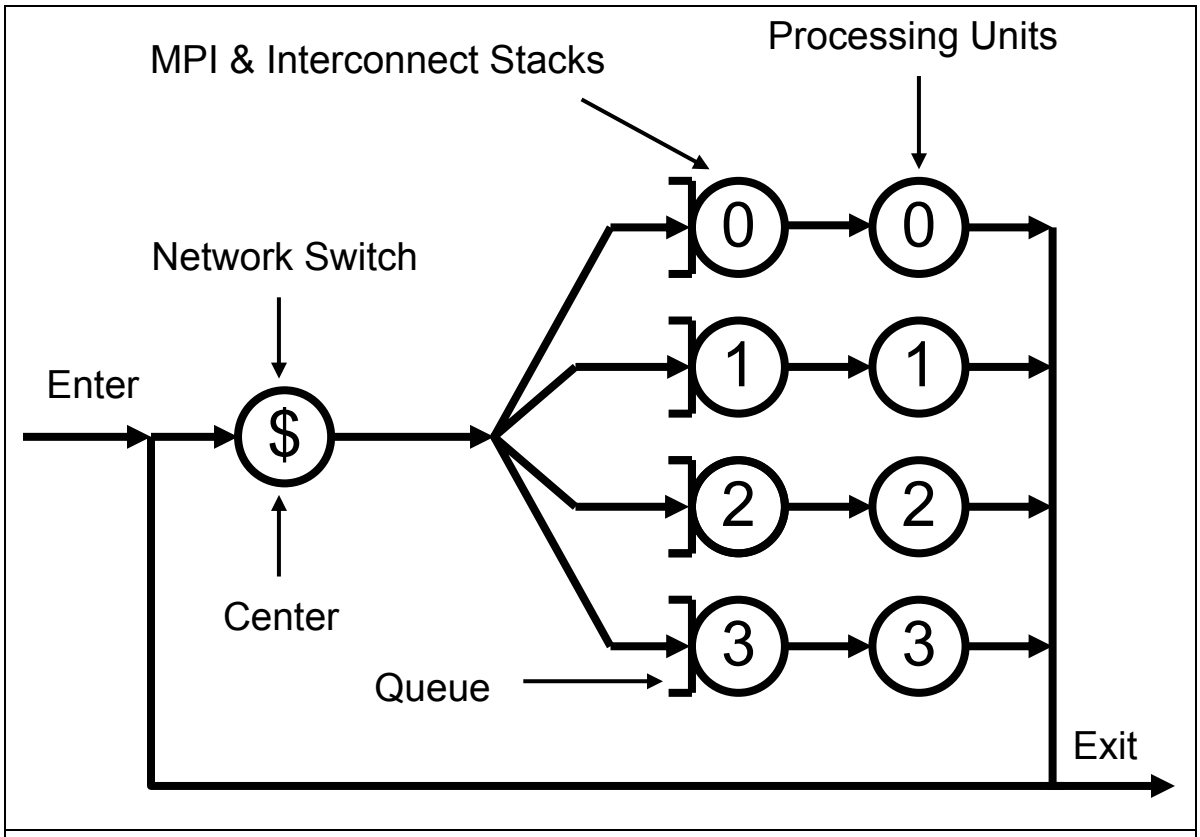

Figure 2 – Cluster Computer QNM Parameterization

Figure 2 repeats Figure 1 from Chapter I, replacing the simple example parameters with ones more appropriate for modeling parallel computers. The aisles in our fictional store are now the fabric of the network switch. The checkout lines and cashiers become our MPI and interconnect stacks, which reflect the fact that messages may be queued waiting for service by MPI. The processing nodes replace our baggers. Finally, instead of customers, we now have messages that cycle through the system. Our cluster is a closed system, meaning there are a maximum number of messages that may be in the system at any time, and new messages must wait for old messages to exit if the system is at capacity. The number of messages is the number of processing units in the system, which models the assumption that a given processor is either generating a message to

send, waiting for receipt of a message to, receiving a message, or computing because of a received message. Thus, messages become the basic unit of work in the model.

The algorithm used to solve the QNM model in the above description, is the Mean Value Analysis (MVA) algorithm, given in Equation 1 below. In the equation, the number of centers (K) is the sum of the number of computation nodes, plus one network switch, plus one computation delay center. While Figure 2 above shows a computation delay center for each queueing center, since each delay center adds the same constant average delay to a customer, using one delay center through which all messages must pass produces equivalent results as including a delay center for each queueing node. The number of customers in the system (N) assumes one message per node, and is equal to the number of computation nodes. The service demand (D_k) for the switch was originally number of messages divided by bandwidth plus latency (though, as explained later in Chapter 3, Section C2, the service demand was explicitly set to zero in some cases). D_k for the processing nodes was the amount of time it took the average node to complete its (non-MPI) calculations Finally, D_k for the queueing nodes (the MPI stack) was the amount of time MPI was active during message transfer. Upon solution, the model provides the system throughput (X), the residence time for an average message at each center (R_k), and the average queue length for each queueing center (Q_k).

The first line of the algorithm in Equation 1 initializes the queue lengths for each queueing center to zero. The second line iterates the algorithm over each customer in the system. The third line iterates over each service center, calculating the residence time at that center for the number of customers currently determined by the previous loop on the second line. For delay centers, this simply adds a delay for that customer as it receives

service. For queueing centers, it adds a delay based on the number of customers waiting for service at that center and the service time for that center. The fourth line determines the current system throughput given the delays calculated on line three and the number of customers determined on line two. The final, fifth line, then updates the queue lengths at each center given the current system throughput and residence times as calculated by lines three and four. Once all iterations are complete, X, R_k, and Q_k as defined in the preceding paragraph, are returned to the user

$$\text{for } k \leftarrow 1 \text{ to } K \text{ do } Q_k \leftarrow 0$$

$$\text{for } n \leftarrow 1 \text{ to } N \text{ do}$$

$$\text{for } k \leftarrow 1 \text{ to } K \text{ do } R_k \leftarrow \begin{cases} D_k & \text{(delay centers)} \\ D_k\left(1+Q_k\right) & \text{(queueing centers)} \end{cases}$$

$$X \leftarrow \frac{n}{\sum_{k=1}^{K} R_k}$$

$$\text{for } k \leftarrow 1 \text{ to } K \text{ do } Q_k \leftarrow XR_k$$

$$\boldsymbol{K \equiv \text{Number of centers}}$$
$$\boldsymbol{Q_k \equiv \text{Queue length at center } k}$$
$$\boldsymbol{N \equiv \text{Number of customers in the system}}$$
$$\boldsymbol{R_k \equiv \text{Residence time at center } k}$$
$$\boldsymbol{D_k \equiv \text{Service Demand at center } k}$$
$$\boldsymbol{X \equiv \text{System Throughput}}$$

Equation 1– Single Class Mean Value Analysis Mathematical Algorithm[31]

[31] [Lazowska, 1984]

III – Methodology and Experimental Environment

This section begins by describing and defining the experimental systems and software in Sections A and B, and continues by describing the experimental procedure used in Section C. Sections D, E, and F describe some of the pre-experimental data analysis that was necessary for result analysis.

A – Experimental Systems

1 – Keck Cluster

The Keck Cluster is the University of San Francisco Department of Computer Science's 24.67 GFlop supercomputer. As described on its website, the Keck Cluster is "… a 64 node Beowulf cluster … [containing] Dual Pentium III 1GHz CPUs, 1GB RAM, [and a] Myrinet Network card … connected by … a 2Gbps Myrinet network used exclusively for communication between MPI programs."[32] "Beowulf Clusters are scalable performance clusters based on commodity hardware, on a private system network, with open source software (Linux) infrastructure."[33]

The default MPI environment on the Keck Cluster is Myrinet's MPICH-GM v. 1.2.4..8a, which was used to access all compilation, linkage, and execution utilities.[34] The Myrinet hardware is version LANai 9, PCI64B. See Chapter VII-A-1 for bandwidth and latency details.

The Keck Cluster is a login/logout system in which the user explicitly reserves the desired nodes for exclusive use and for an indefinite period. Hence, it has no batch

[32] [Keck, 2004a]
[33] [Beowulf, 2005]
[34] [Keck, 2004b]

execution control. It runs RedHat Linux 8.0. The University of San Francisco replaced the Keck Cluster and all its supporting documentation with a new cluster in the fall of 2006.

2 – MCR

MCR is a multiple node supercomputer located at Lawrence Livermore National Laboratory. MCR stands for Multiprogrammatic Capability Cluster.

MCR is "a large (11.2 TF) tightly coupled Linux cluster … has 1,152 nodes, each with two 2.4GHz Pentium 4 Xeon processors and 4GB of memory … runs the LLNL CHAOS software … which incorporates … Red Hat Linux."[35] Its peak performance is 11.06TFlop/s.[36]

Compilation, linkage, and execution was performed on MCR using the native Intel compilers icc and ifort, both v. 8.1, under both CHAOS v. 2 and CHAOS v. 3.

MCR uses a Quadrics QsNet Elan 3 interconnect, which delivers high bandwidth (>300 MB/s) with low latency (<5.0 μs).[37] MCR utilizes the LCRM/SLURM batch control system, and Quadrics MPI, a derivative of MPICH 1.2.4.[38]

3 – ALC

ALC is another multiple node supercomputer located at Lawrence Livermore National Laboratory. ALC stands for ASC Linux Cluster.

[35] [M&IC, 2004]
[36] [LCOCF, 2006]
[37] [M&IC, 2002]
[38] [Linux, 2006]

ALC has 960 nodes, each with two 2.4GHz Pentium 4 Prestonia processors and 4GB of memory, and runs the LLNL CHAOS software.[39] Its peak performance is 9.2 TFlop/s.[40]

Compilation, linkage, and execution was performed on ALC using the native Intel compilers icc and ifort, both v. 8.1, under both CHAOS v. 2 and CHAOS v. 3.

ALC uses a Quadrics QsNet Elan 3 interconnect, which delivers high bandwidth (>300 MB/s) with low latency (<5.0 μs).[41] ALC utilizes the LCRM/SLURM batch control system, and Quadrics MPI, a derivative of MPICH 1.2.4.[42]

ALC and MCR are very similar systems in hardware, installed software, and configuration. The main differences lie in the concrete hardware components (motherboards, chipsets, etc.). Thus, they provide similar, though not identical, test systems, and perform similarly for the same application.

B – Experimental Software

1 – NAS Parallel Benchmarks

The flavor used for the benchmarking in the study is NPB 2.4, which uses MPI, and is commonly used by other researchers.

We chose the CG test as the base test for collecting measurement data about the test systems, in part because of its common use for benchmark studies. CG's main loop:

- Post non-blocking point-to-point receive to nodes containing neighboring data.

[39] [ASC, 2004]
[40] [LCOCF, 2006]
[41] [LCOCF, 2006]
[42] [Linux, 2006]

- Perform blocking point-to-point send.
- Wait for receive to complete.
- Perform calculation and loop.

provides behavior typical of many MPI programs, is easy to measure, and easy to model.

2 – MpiP

As described in its documentation, "mpiP is a lightweight profiling library for MPI applications."[43] The LLNL staff developed mpiP, and it is a publicly available resource, through either LLNL or SourceForge.

MpiP intercepts an application's linkage to MPI programs using the standardized PMPI interface, thus allowing mpiP to collect information concerning a variety of MPI calls. The application calls the mpiP routines, which then collect some system state data, such as the call stacks and procedure call timings, and then call the related MPI routines. When MPI routines return, mpiP records timing and counter information. MpiP can be linked at run-time, thus avoiding the need to recompile. Generating global statistics only at the end of execution, mpiP has very low overhead. This is explored further in Chapter III, Section F.

MpiP maintains several control settings, manipulated by setting system variables, passing options on the command line, or during program execution. By default, mpiP begins timing as soon as the MPI_Init() routine is encountered in the code. However, the NPB provide for one warm-up iteration of the code to ensure the necessary data is in memory and the cache is full. In order to accommodate this warm-up iteration in mpiP

[43] [MpiP, 2005]

and provide consistency in the measured timings, the program began execution with the mpiP timers disabled. The NPB code was modified such that when NPB began timing code execution, the mpiP timers were also enabled and began data collection.

3 – NPB Spreadsheet

The NPB spreadsheet, developed as part of an LLNL Research Subcontract, is designed to receive, as input, selected values from the NAS-PB Suite and mpiP output files. It then uses these values to calculate model inputs for the QNM Solver, as well as generating the command line for inMaker, a program written for this project that generates the QNM Solver input file for that dataset.

The spreadsheet also performs error analysis between modeled and measured values, and graphically displays the results, along with a breakdown of the measured components and their resultant model outputs. It also graphically compares the components of the model's wall clock time for the application to the measured components of the wall clock time. Additional technical details about the NPB spreadsheet are available in ancillary documentation.

4 – MpiPfilter

MpiPfilter is a simple Java program, developed as part of an LLNL Research Subcontract, that filters the output files from mpiP and NAS-PB and creates a text file usable as input for the NBP Spreadsheet.

The program reads the output files generated by NPB and mpiP and compiles data on several metrics produced by NBP, mpiP, the Linux time command, or some combination of these. These include: the number of messages generated, average size of messages,

aggregate application, MPI and MPI wait times, elapsed time, MOPS/s, MOPS/s/process, and, when possible, CPU utilization and average elapsed time. We collected some unused metrics for future research. In addition, some identifying metadata, such as date run and head node PID to assist in separating the various runs. Using the average message size and internally programmed data tables, the program also calculates the network bandwidth and latency. Results are returned as a data file for entry into the NPB Spreadsheet.

5 – QNM Solver

The QNM (Queueing Network Model) Solver is a Java program, ported from algorithms and FORTRAN code in [Lazowska, 1984], and developed as part of an LLNL Research Subcontract. Using input files generated by inMaker, based on values from the NPB spreadsheet, gathered from mpiP and NAS PB suite files, the program models the system as a queueing network. The solver performs single class mean value analysis (Equation 1 above), multiple class MVA, single class load dependent service center solution, and is capable of batch execution. We developed the new solver to allow easy parameterization and alteration of the modeling software in order to accommodate the peculiarities of the modeled environment.

The output of the solver is entered into the NPB spreadsheet to complete the modeled vs. measured validations.

6 – InMaker

A simple Java program, developed as part of the LLNL Research Subcontract, which, when given parameters for the QNM solver, will create as output a file that can be used

as input for the QNM solver. This simplifies execution of the QNM solver program by freeing the user from the repetitive task of entering service center parameters.

7 – LBW

LBW is a latency and bandwidth tester developed at LLNL. According to the documentation, LBW "…attempts to measure the point-to-point message passing latency and bandwidth. The test uses two MPI processes that repeatedly exchange messages."[44] LBW is configurable in the number of messages passed, message length, and whether the communication is synchronous (blocking), or asynchronous (non-blocking). More details of our LBW data collection are given in Appendix A.

C – MPI Performance Measurement and Modeling Procedure

1 – Data Collection and Measurement

The methods shown in Figure 3 and described subsequently were applied to the collection and analysis of data from the various experimental systems:[45]

[44] [Faulkner, 2005]

[45] Ovals represent entry and exit points from the control flow. Rectangles represent procedures that are nearly identical for each test machine. Diamonds represent procedures that may require customization for each machine. Circles represent discontinuities in the graph, continue reading from corresponding symbol in circle.

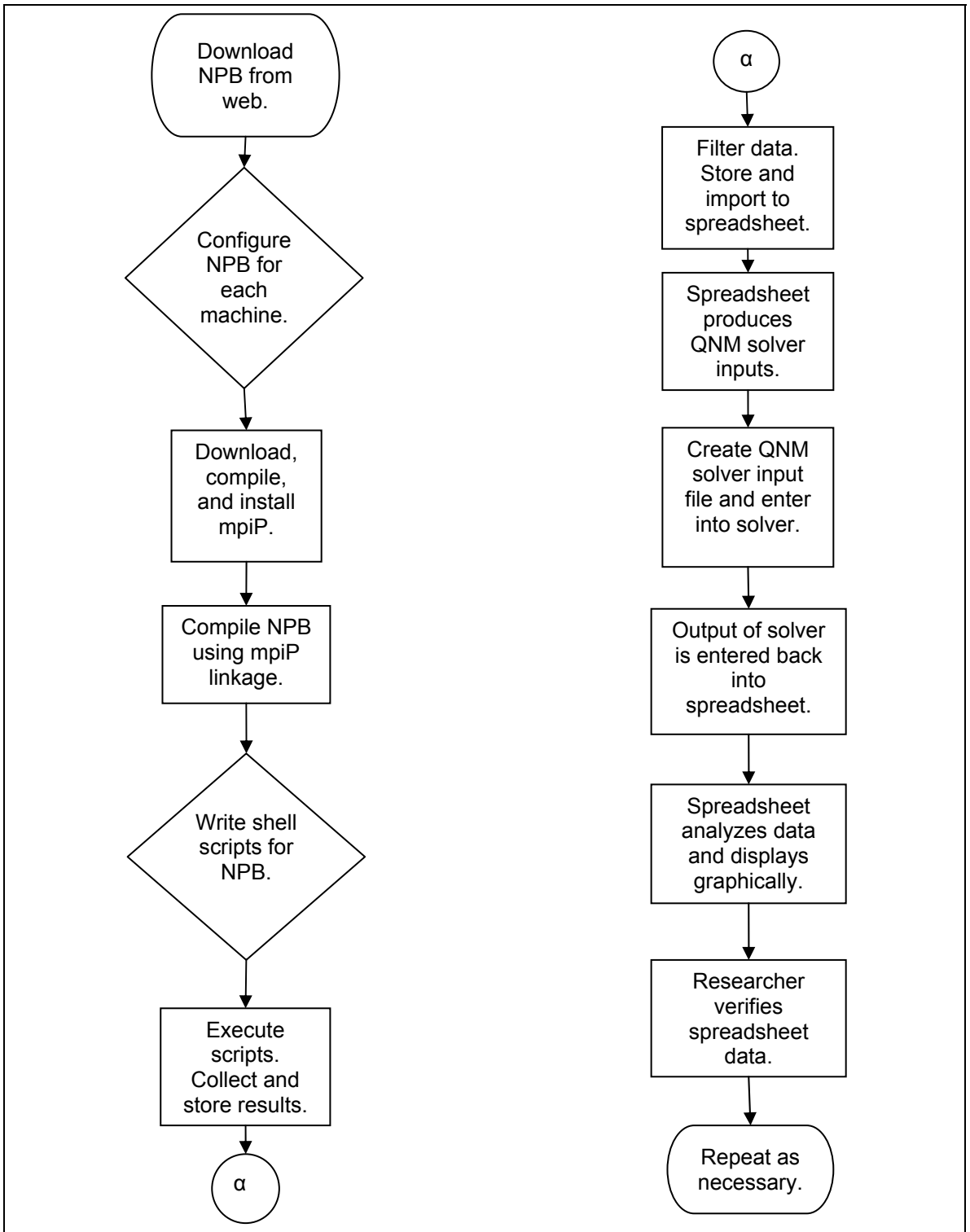

Figure 3 – Measurement and Modeling Control Flow

- Downloaded the NAS Parallel benchmarks from the NAS website.[46]
- Configuration files for the NAS PB were modified for each machine. Minor errors in benchmark code were corrected to prevent compiler errors and select the proper NPB timing routines, as provided by NAS.
- Downloaded, compiled, and installed mpiP from LLNL website.[47] [48]
- The NAS PB executables were compiled using mpiP linkage to provide detailed analysis of MPI calls and timings.
- Shell scripts were written for each suite of NAS PB tests to provide proper environment setup and to ease execution.
- The shell scripts were executed, and the resultant data files containing MPI call and timing information were captured and stored in a directory.
- The mpiPfilter program was run on the data to extract relevant timing and call information, as well as ease data file parsing.
- The resultant output files were also stored and imported into the NPB spreadsheet for storage and ease of calculation.
- Calculations in the NPB spreadsheet produced inputs for the QNM solver using imported data from the data files above, and created prototype execution commands for inMaker.
- QNM solver input files were created using inMaker, and fed to the solver using single class MVA batch mode.

[46] http://www.nas.nasa.gov/Software/NPB/
[47] http://www.llnl.gov/CASC/mpip/
[48] MpiP is also available through SourceForge.

- Outputs from the QNM solver were copied back into the NBP spreadsheet, which performed error analysis on the modeled vs. measured inputs and graphically displayed the results.
- The final NPB spreadsheet was then inspected for continuity, alignment of data, error, and other anomalies to ensure the spreadsheet was functioning properly, that all necessary data was collected, and that all application executions terminated properly.
- As necessary, the model was corrected to ensure the above states held true and the QNM solver was rerun on the newly corrected data.

2 – QNM Model and Mean Value Analysis

The QNM models were solved using the single class mean-value analysis, as shown in Equation 1, which is easy to parameterize and eliminates complexity due to multiple classes of messages and load dependent servicing times.

Parameters for the QNM model were determined as follows:

- The number of centers is the number of nodes allocated to solving the problem, plus two. The additional nodes represent the switch and the time spent on computation in the CPU. The switch and CPU are parameterized as delay centers, as they generally do not queue messages for servicing, whereas the centers representing the MPI message servicing nodes are parameterized as queueing centers, as they may have multiple messages waiting on servicing.
- The number of customers is equal to the number of computational nodes in the system.

- The delay for the switch (when used) is determined by dividing the average message size by the bandwidth passing through the system and adding the latency. This approximation avoids the complexity of having multiple classes of servicing for various messages sizes. For our purposes, this value was unused and explicitly set to zero, as this produced better comparison between the modeled and measured results. What switch delay does exist is included in the service time for the queueing centers (i.e. the MPI and interconnect stack). Using a nonzero switch delay center results in counting the switch delay twice in the baseline model.

- The delay for the CPU is the total application time (TTS) minus the total time in MPI calls, with the result divided by the number of messages in the system.

- The aggregate service demand for the queueing MPI nodes is the total time spent in MPI calls minus the time spent in MPI_Wait. The product of the number of computational nodes and the number of messages then divides this value in the system to produce the input for the MVA algorithm. Since the NPB CG benchmark only uses point-to-point communication, MPI_Wait captures all the explicit wait time.

- The MVA was solved for queue length, residence time, and throughput.

D – Correlating NAS-PB CG Classes to Numeric Problem Sizes

Because serial TTS (i.e. execution time for a single processor) could be determined for NPB, and is a reasonable work metric (i.e. measure of program size),[49] we attempted to find parameters in the NPB output files that would provide for a similar work metric that was machine independent. We did this because we were unable to determine an appropriate work metric from the NPB input files. As a step toward our work metric, we noted that message size and number of messages remained constant for a given allocation of processors for a given problem class, as was expected. Trial and error search then gave us the work metric in Equation 2 below.

Class/ # P	S	W	A	B	C	D
1	8	8	8	8	8	
2	2,776	13,862	27,720	148,557	297,109	
4	2,776	13,862	27,720	148,557	297,109	
8	1,191	5,937	11,870	63,587	127,169	
16	1,191	5,937	11,870	63,587	127,169	1,271,675
32	558	2,772	5,540	29,661	59,318	593,138
64	558	2,772	5,540	29,661	59,318	593,138
128	271	1,335	2,665	14,259	28,513	285,084
256	271	1,335	2,665	14,259	28,513	285,084
512	135	653	1,302	6,953	13,900	138,957

Table 1 – Message Sizes for NPB CG

[49] [Grama, 2003]

Class/ # P	S	W	A	B	C	D
1	1	1	1	1	1	
2	3,152	3,152	3,152	15,752	15,752	
4	6,304	6,304	6,304	31,504	31,504	
8	22,088	22,088	22,088	110,408	110,408	
16	44,176	44,176	44,176	220,816	220,816	294,416
32	126,272	126,272	126,272	631,232	631,232	841,632
64	252,544	252,544	252,544	1,262,464	1,262,464	1,683,264
128	656,768	656,768	656,768	3,283,328	3,283,328	4,377,728
256	1,313,536	1,313,536	1,313,536	6,566,656	6,566,656	8,755,456
512	3,233,792	3,233,792	3,233,792	16,166,912	16,166,912	21,555,712

Table 2 –Number of Messages for NPB CG

Class/ # P	S	W	A	B	C	D
2	24.284E+9	605.671E+9	2.422E+12	347.632E+12	1.39E+15	
4	48.569E+9	1.211E+12	4.844E+12	695.265E+12	2.781E+15	
8	31.343E+9	778.644E+9	3.112E+12	446.414E+12	1.786E+15	
16	62.686E+9	1.557E+12	6.224E+12	892.828E+12	3.571E+15	476.117E+15
32	39.373E+9	970.474E+9	3.875E+12	555.35E+12	2.221E+15	296.097E+15
64	78.746E+9	1.941E+12	7.75E+12	1.111E+15	4.442E+15	592.194E+15
128	48.263E+9	1.171E+12	4.666E+12	667.552E+12	2.669E+15	355.792E+15
256	96.526E+9	2.342E+12	9.331E+12	1.335E+15	5.339E+15	711.583E+15
512	58.763E+9	1.381E+12	5.481E+12	781.499E+12	3.124E+15	416.218E+15
avg	54.284E+9	1.329E+12	5.301E+12	759.149E+12	3.036E+15	474.667E+15

Table 3 –Work Metric for NPB CG

$$WM = MessageSize^2 \cdot NumberMessages$$

Equation 2– Formula for Parameter-Based Work Metric

Table 1, Table 2, and Table 3 above show the message sizes, number of messages and new work metric for NPB CG. Equation 2 shows how the new work metric is derived.

Next, we plotted the serial runtimes and the new work metric together and examined the tracking. While the work metric does exhibit some waviness, the resultant lines follow a near-horizontal linear trend around the average. Spacing between the classes for the new work metric was also consistent with spacing between the classes for serial runtime. This is shown graphically in Figure 4 below. Because the new work metric

seemed so consistent with time to solution, and because it is machine independent, we adopted the average work metric for each class to replace time to solution. One drawback of this process is that it required actually running the code to collect the data. However, we did this as a matter of expediency, and detailed program analysis, possibly done during coding, could eliminate this need.

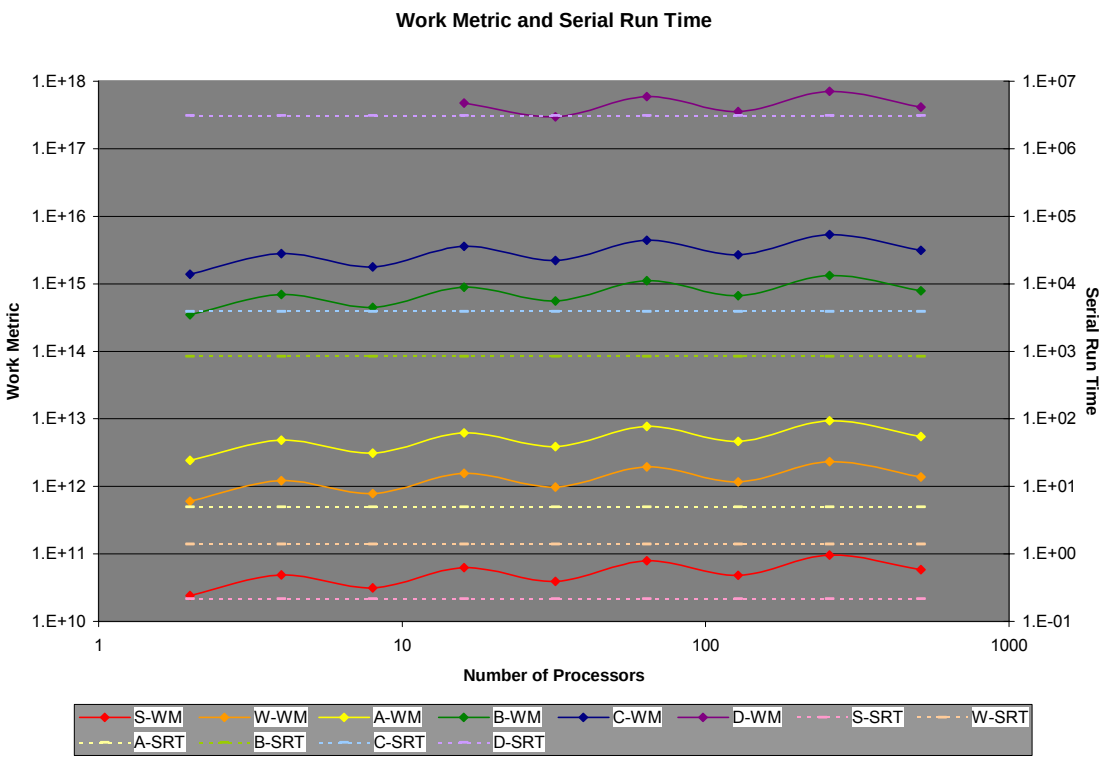

Figure 4 – Work Metric and Serial Run Time

E – Determining Bandwidth and Latency

The first attempts at modeling utilized previously published bandwidth and latency data.[50] However, the data being utilized were for synchronous communication models (where a blocking send is followed by a blocking receive), whereas NPB-CG utilizes an asynchronous communication model (a non-blocking receive, followed by a blocking

[50] [Faulkner, 2005]

send, then a wait for the receive to complete). The LBW test was run on each machine to capture latency and bandwidth values for inter/intra-node and synchronous/asynchronous communications, as shown in Appendices A-2 and A-3. Configuration for LBW is as follows:

- All tests contained the "-a" switch to obtain information for each MPI process.
- All tests were submitted to the batch partition requesting two nodes, one processor per node active on our problem.
- For both bandwidth testing ("-B" switch) and latency testing ("-L" switch):
    - Buffer sizes ("-b nnn" switch) assumed these values: 40, 400, 1,000, 10,000, 50,000, 100,000, 500,000, 1,000,000, 2,500,000, 5,000,000, 10,000,000, 15,000,000, and 20,000,000.
    - All buffer sizes were tested in both synchronous and asynchronous modes ("-s sync/async").
    - Test was repeated ("-n nnn") so that the time for each run was approximately 5 min. The repetition values were 15,000,000, 7,500,000 (x2), 5,000,000, 1,750,000, 700,000, 100,000, 45,000, 20,000, 7,500, 4,000, and 2,500.

We ran each test on five separate occasions, with a minimum of 48 hours between runs, in order to provide a random "typical" machine configuration. The mean value of

the five runs for a given message size and communication model was determined, and these values were utilized for the machine's bandwidth and latency.[51]

To guard against the possibility of "outliers" in the LBW testing runs, we compared the mean-over-five value mentioned above to the mean-over-three-of-five, where we removed the highest and lowest values before calculating the mean value. We noted no significant differences between these calculations, and accepted the mean-over-five as being easiest to implement. Additionally, we also re-ran any test run where the shape of the curve of the plotted data varied significantly from previous behavior (additional peaks, valleys, or plateaus) as aberrant; such behavior is not what the average user under normal operating circumstances would typically see. While it obviously is possible for the system to produce such behavior, such an abnormal state could not be long maintained, and is thus discounted and more appropriate results (where the shape of the curve did not vary significantly) used instead.

The resultant bandwidth and latency data provide values that are typical for each machine under normal operating conditions, and most closely approximate behavior the average user would expect. For more detailed analysis and results, see Chapter VII, Section A.

F – Determining mpiP Overhead

In order to determine the overhead associated with mpiP, the LBW test mentioned in Section E above was run both with and without linkage to the mpiP libraries, using the

[51] The LBW tests could not be run on the Keck Cluster, and are not considered in the following section.

default MPI and mpiP configurations. Using Equation 3 below, we calculated and examined the percentage of mpiP overhead.

$$\%OH_{mpiP} = \frac{LBW_{mpiP} - LBW_{no-mpiP}}{LBW_{no-mpiP}}$$

Equation 3 – Determining mpiP Overhead

As seen in Table 4, Table 5, Figure 5, Figure 6, Figure 7, and Figure 8, for large messages (those over 10,000 bytes) mpiP on MCR has less than 2.25% overhead for bandwidth and less than 2.5% overhead for latency in the inter-node asynchronous cases, which most closely resemble the behavior of our test benchmark. Overhead on ALC is even less, with large messages having less than 1.6% overhead for bandwidth and less than 0.8% overhead for latency. These differences, though interesting, are secondary to the focus of the current research and are reserved for future exploration. However, for small messages (those under 10,000 bytes) mpiP overheads become quite significant, in some cases exceeding 50%.

We collected no data for the Keck Cluster, as we could not compile or execute the LBW routines in the available time. However, as the Keck Cluster architecturally is similar to both ALC and MCR, it is reasonable to assume we would obtain similar results.

Message Size	Bandwidth (10e6 B/s)				Latency (µs)			
	Intra		Inter		Intra		Inter	
	Sync	Async	Sync	Async	Sync	Async	Sync	Async
40	-20.21%	-7.69%	-17.40%	-31.56%	26.41%	11.49%	27.35%	44.05%
400	-11.54%	-1.75%	-13.06%	-26.18%	15.57%	2.49%	20.22%	32.63%
1,000	-11.48%	-3.76%	-11.74%	-16.75%	13.94%	2.61%	17.86%	18.43%
10,000	-6.83%	-11.47%	-1.48%	-10.22%	8.92%	12.64%	7.35%	10.34%
50,000	-2.78%	-3.38%	4.10%	-1.99%	6.24%	5.34%	1.79%	2.37%
100,000	-7.28%	-3.54%	4.63%	-2.22%	13.87%	9.52%	1.57%	1.22%
500,000	1.95%	0.52%	5.46%	-1.96%	1.34%	-0.73%	0.75%	0.35%
1,000,000	2.01%	1.06%	5.58%	-1.60%	-0.12%	-0.85%	0.67%	0.39%
2,500,000	2.80%	1.16%	5.61%	-1.92%	-0.31%	-0.68%	0.58%	0.26%
5,000,000	2.02%	1.03%	5.62%	-2.08%	-0.59%	-0.54%	0.45%	0.19%
10,000,000	2.01%	1.27%	5.63%	-2.22%	-0.23%	0.29%	0.37%	-0.05%
15,000,000	2.97%	1.59%	5.70%	-2.07%	-0.77%	1.36%	0.40%	-0.12%
20,000,000	3.22%	1.24%	5.72%	-1.78%	-1.47%	-1.30%	0.39%	-0.14%

Table 4 – mpiP Overhead on MCR

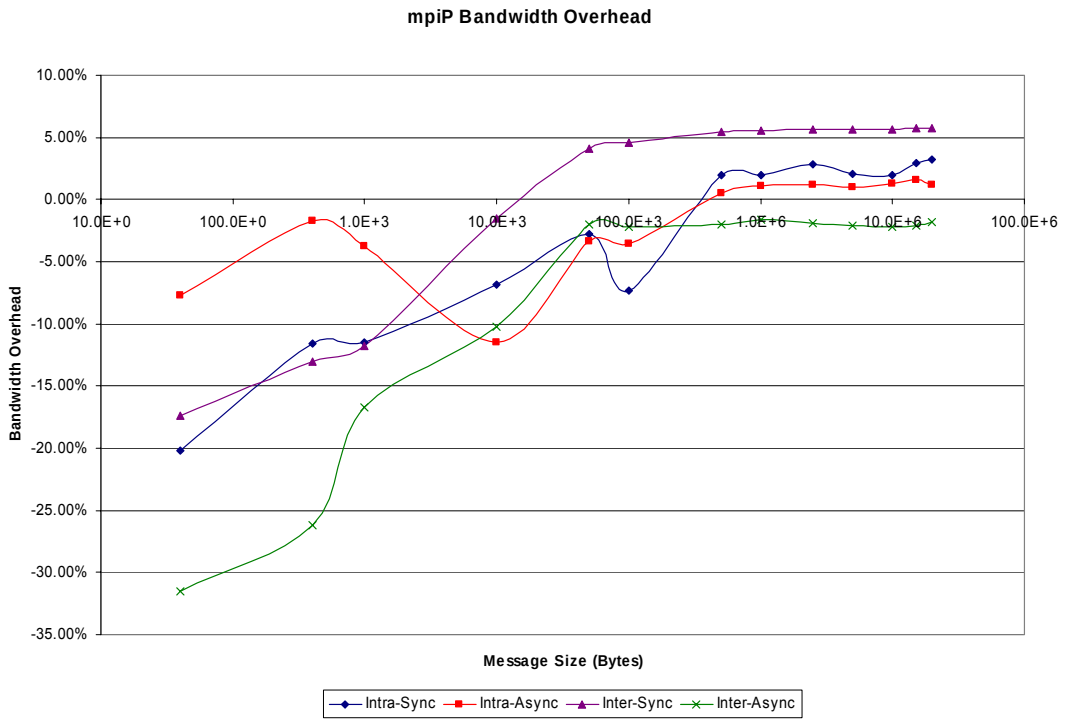

Figure 5 – mpiP Bandwidth Overhead on MCR

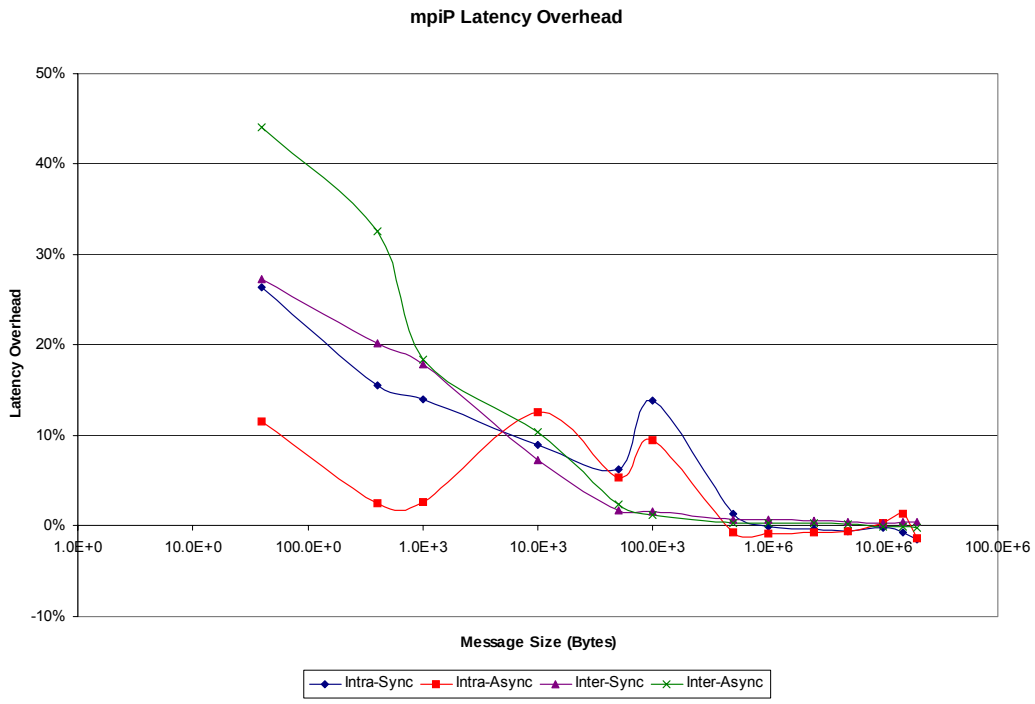

Figure 6 – mpiP Latency Overhead on MCR

Message Size	Bandwidth (10e6 B/s)				Latency (µs)			
	Intra		Inter		Intra		Inter	
	Sync	Async	Sync	Async	Sync	Async	Sync	Async
40	-21.09%	12.33%	-22.44%	-33.24%	24.12%	-11.99%	28.42%	50.09%
400	-9.88%	-0.95%	-16.40%	-26.36%	11.81%	1.11%	18.12%	33.17%
1,000	-10.08%	-3.57%	-13.09%	-20.86%	10.68%	2.53%	12.64%	25.09%
10,000	-7.28%	-8.72%	-6.40%	-6.18%	7.00%	9.93%	4.64%	7.42%
50,000	-2.50%	3.33%	-1.27%	-1.54%	2.20%	3.50%	0.14%	0.55%
100,000	-8.29%	-3.54%	-2.10%	-0.92%	-1.66%	5.66%	0.21%	0.78%
500,000	-5.21%	-3.83%	-1.26%	0.17%	5.69%	2.74%	-0.28%	-0.27%
1,000,000	-3.59%	-1.59%	0.10%	0.00%	3.79%	1.72%	-1.08%	0.00%
2,500,000	-2.17%	-3.51%	-1.09%	0.14%	4.27%	1.66%	-0.28%	-0.01%
5,000,000	-1.71%	-1.31%	-0.10%	0.32%	1.91%	0.04%	-1.25%	0.24%
10,000,000	-0.62%	-1.20%	0.38%	0.35%	2.21%	0.34%	-1.27%	0.24%
15,000,000	-0.33%	-1.42%	0.32%	0.35%	3.35%	0.68%	-1.27%	0.13%
20,000,000	-0.49%	-0.67%	0.23%	0.35%	4.35%	-0.56%	-1.22%	0.11%

Table 5 – mpiP Overhead on ALC

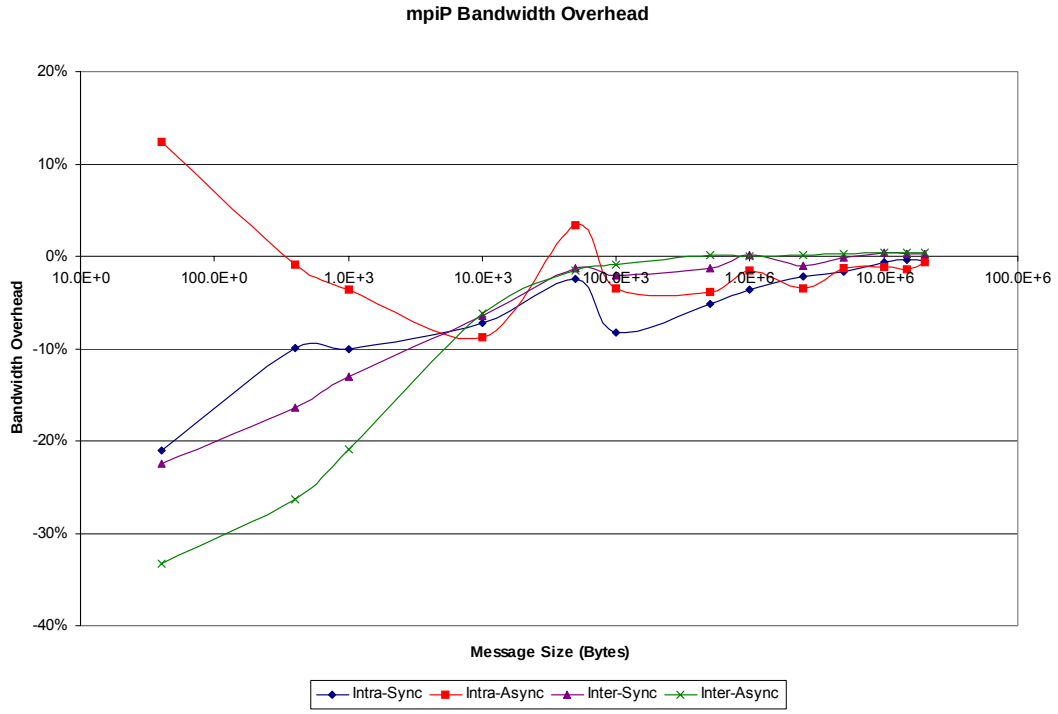

Figure 7 – mpiP Bandwidth Overhead on ALC

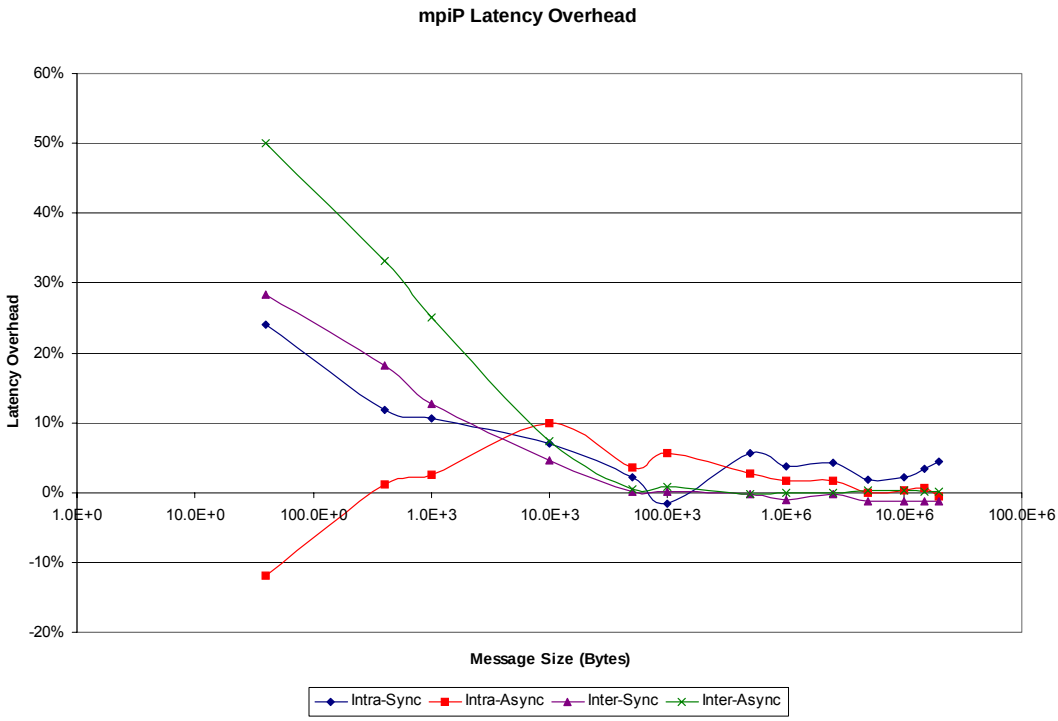

Figure 8 – mpiP Latency Overhead on ALC

Note the complexity of the above graphs. There is no obvious explanation for either the complexity or bumpiness of the results (though one possibility is that inter-node communication is less bumpy than intra-node communication due to caching effects) and, as this was secondary to the focus of the research, was not explored further but left for future research.

IV – Results and Analysis

A – Overview and General Comments

To establish the predictive capabilities of QNM, it was first necessary to establish a mathematical relationship between the various classes (sizes) of problem and the corresponding runtimes for those classes. We outline the technical details for this procedure in Chapter III, Section D. This section is concerned with analysis of the procedure outlined in the previous section, the results of which are contained in Sections B – D below.

We ran sample benchmarks, in particular NPB CG, on all target systems for each class and available processor allocation. We used these results to understand the behavior of actual machines-in-execution, as well as to provide target values for the model. Data were collected apriori to modeling due to the often-lengthy amount of time it took for the experimental benchmarks to schedule and execute on the target machines.

In general, comparisons of the results were good. However, careful examination of the graphical results, shown separately for each machine below, shows unusual behavior for small problem sizes (classes S, W, and A) as the number of processors allocated to solving the problem increases. In these cases, runtime values begin to slip off their previous trend-lines, creating a "hook" in the runtime plot. We explore this effect in more detail in Sections B - D.

Following collection and analysis of actual machine performance, we ran QNM models on the collected data and graphed the data, the results of which are also contained in

Sections B –D below. Comparison of the modeled versus the measured performance is contained in Section E following.

B – MCR

This section contains graphical analysis of the data collected from MCR.

1 –Runtimes With Respect to Class

Note in Figure 9 the points where the application begins to show non-optimal performance. This nadir, at 16 processors for class S, 64 for class W, and 256 for class A, indicates the processor allocation beyond which TTS degrades rather than improves with the addition of more computational nodes.

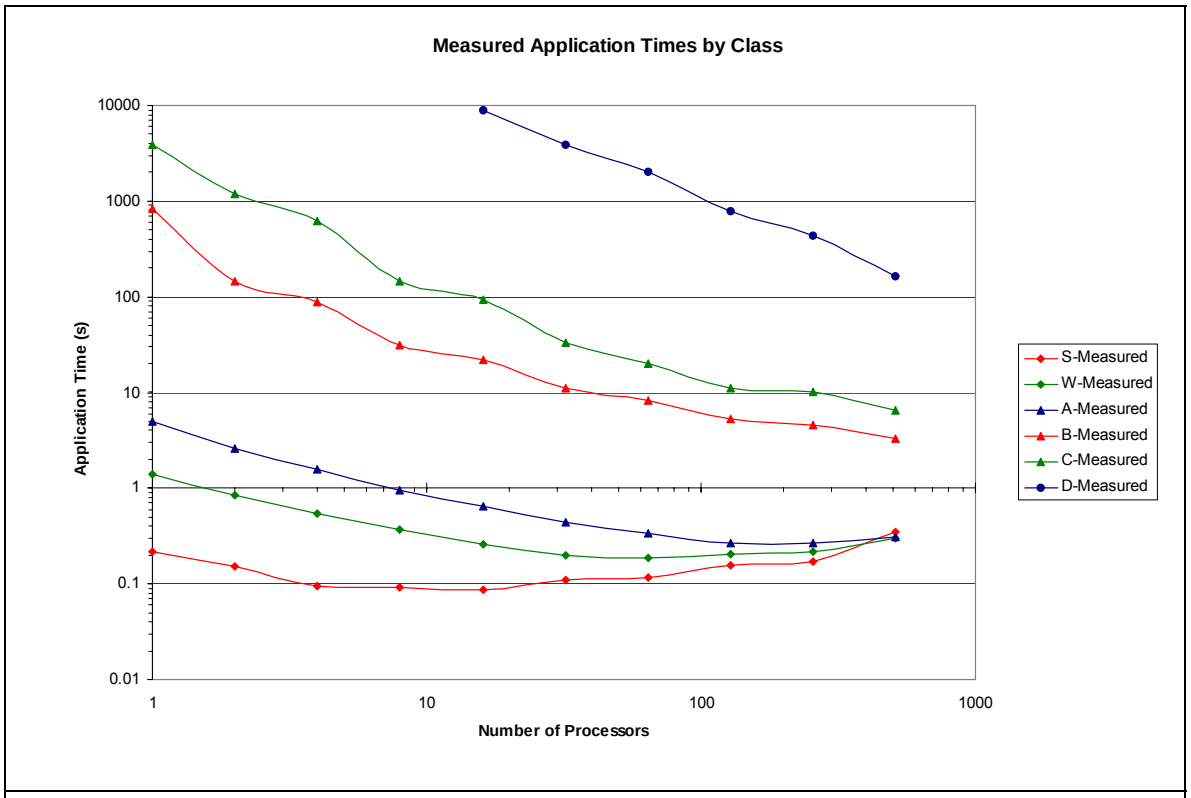

Figure 9 – MCR Measured Runtimes With Respect to Class

Note that Figure 10 is very similar to the plot in Figure 9.

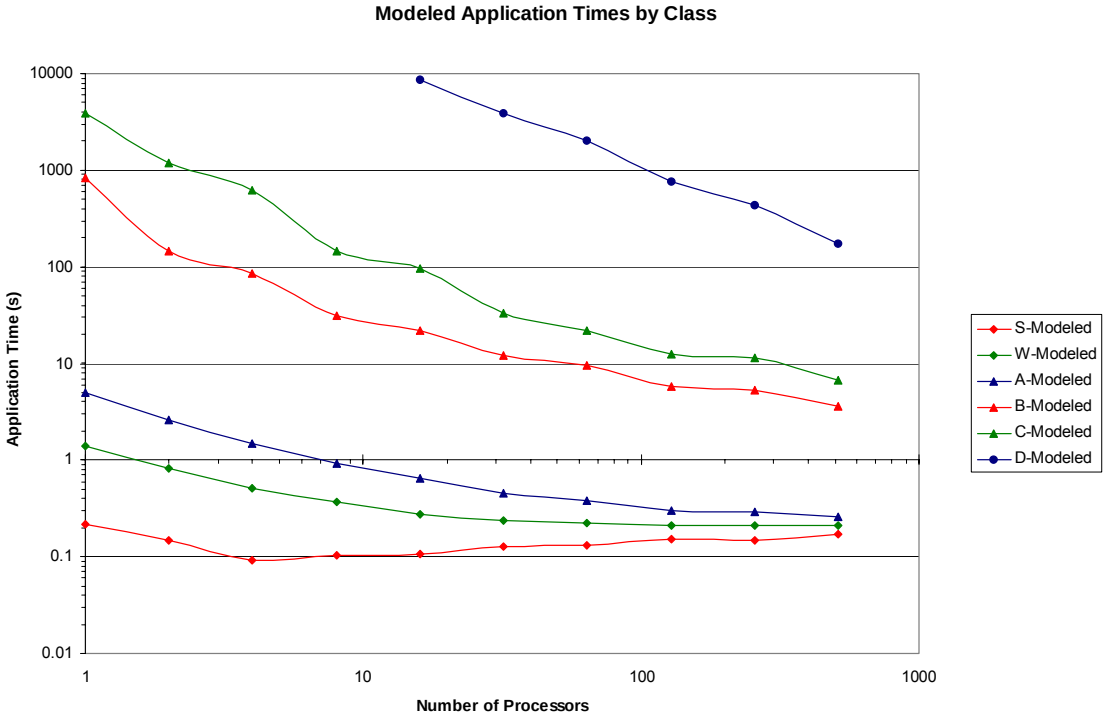

Figure 10 – MCR Modeled Runtimes With Respect to Class

2 –Runtimes with Respect to Processors Allocated

In Figure 11 the horizontal axis is the work metric (Chapter III, Section D), which measures the problem size, from S on the left, to D on the right. Note in Figure 11 the point where the application begins to show non-optimal performance, of a different sort than in Subsection 2.1 above. This nadir, between classes W and A for 512 processors, indicates the processor allocation is in a region which TTS degrades rather than improves as the problem gets smaller.

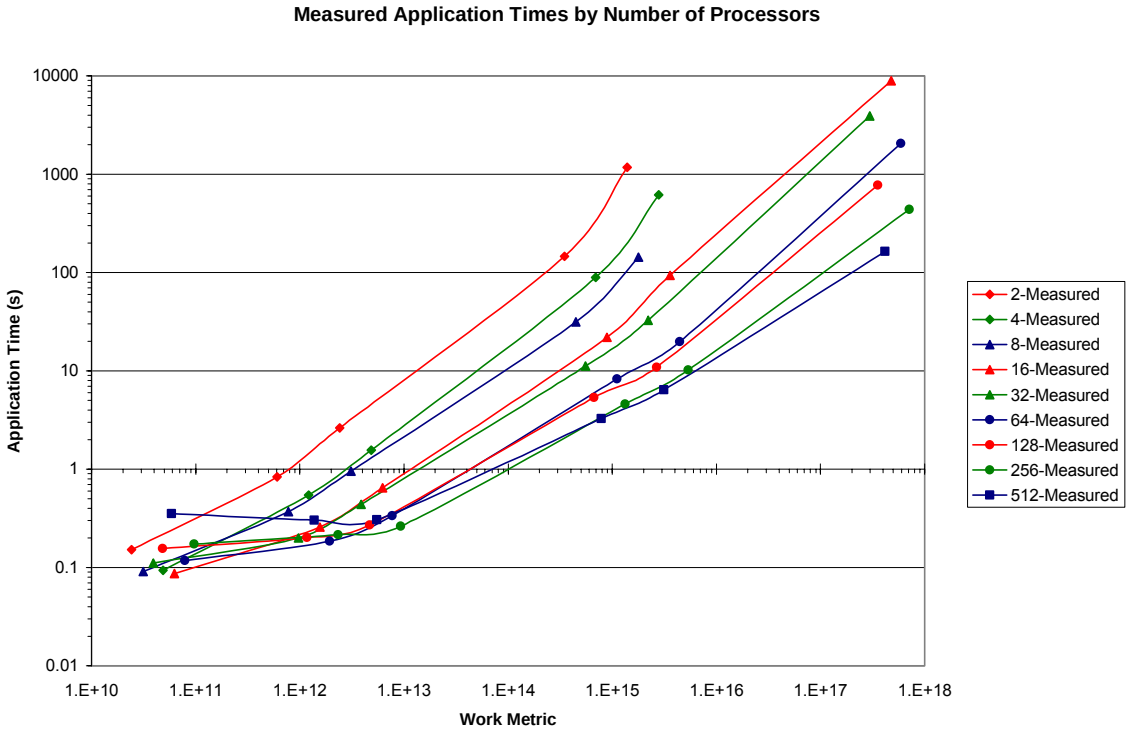

Figure 11 – MCR Measured Runtimes With Respect to Processors Allocated

Note that Figure 12 is very similar to the plot in Figure 11.

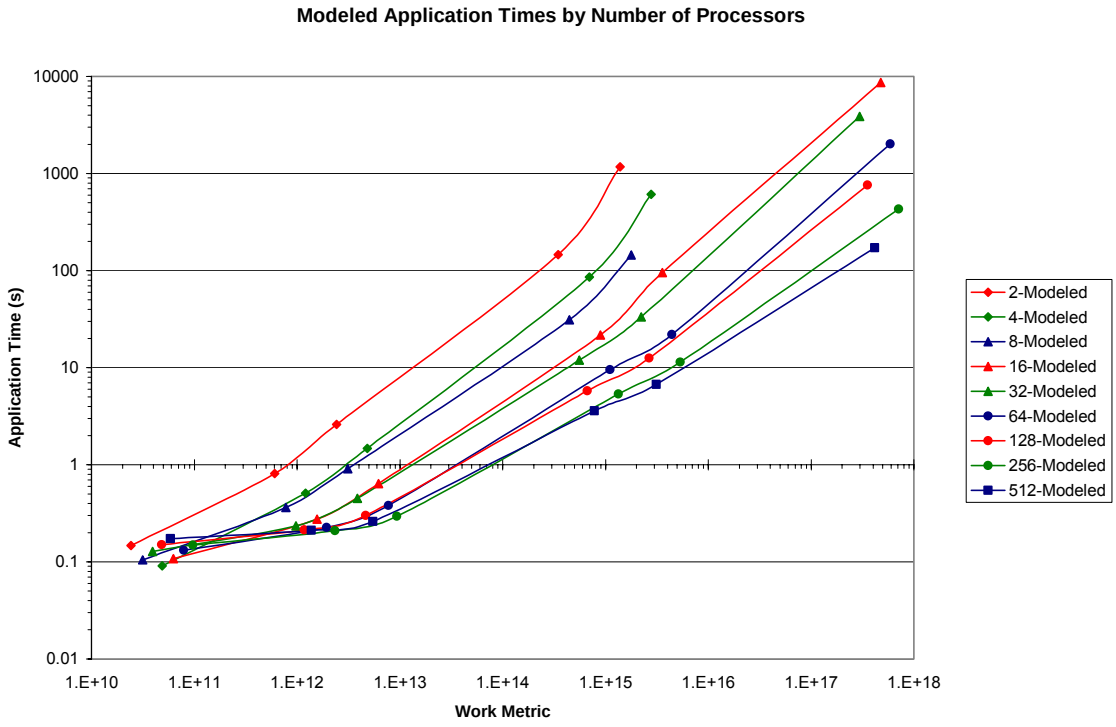

Figure 12 – MCR Modeled Runtimes With Respect to Processors Allocated

3 –Runtimes with Respect to Processors Allocated and Class Size

These graphs are a combination of the above four graphs, and shows the above non-optimal application behavior as valleys along the various contours.

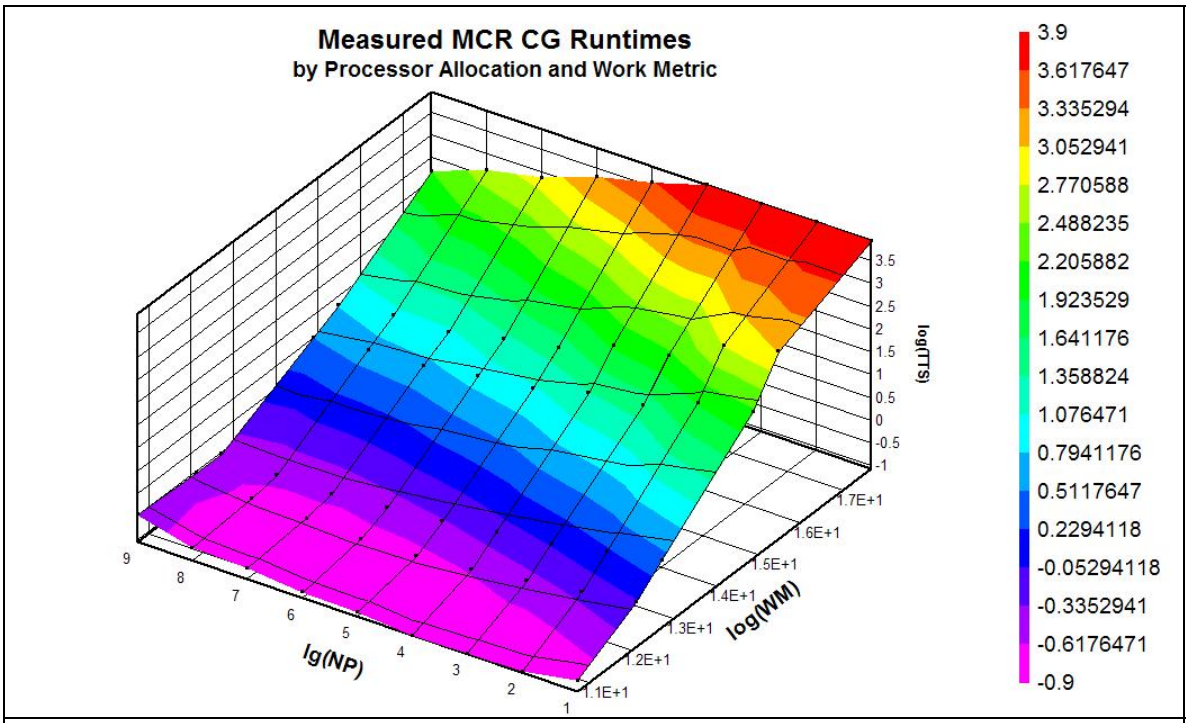

Figure 13 – MCR Measured Runtimes With Respect to Processors Allocated and Class Size

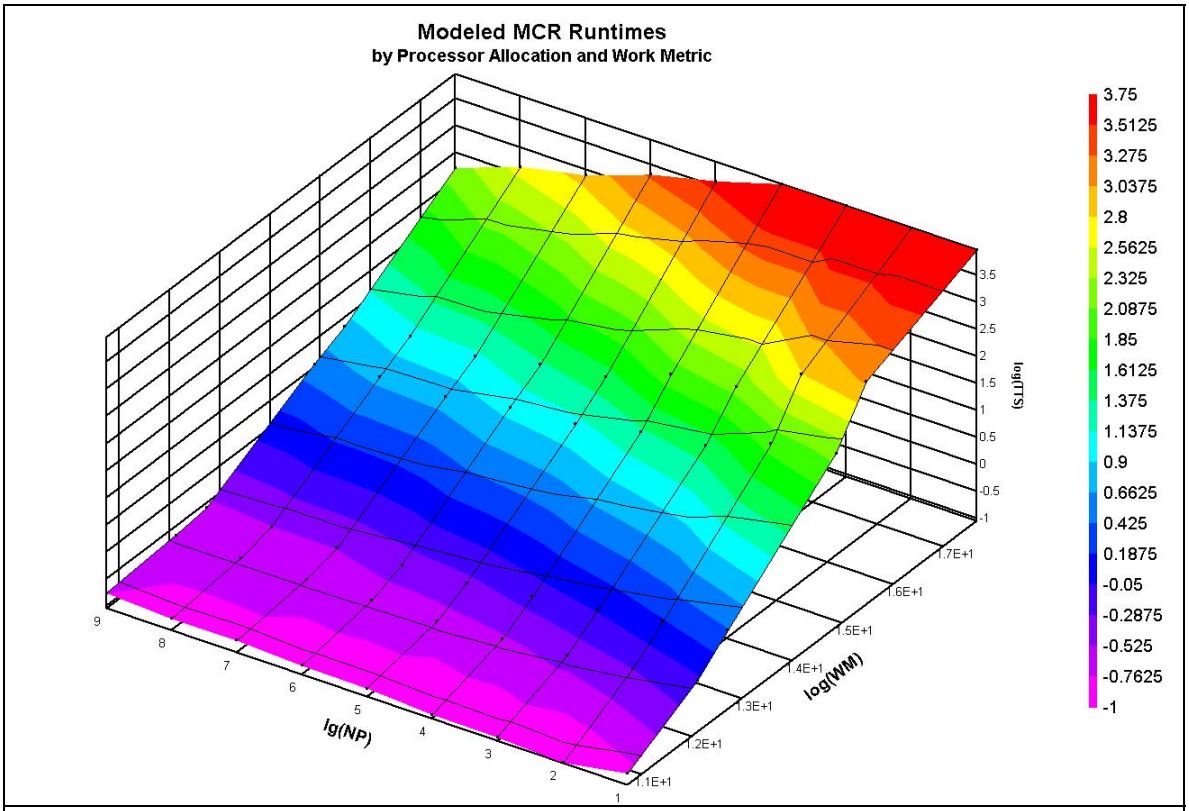

Figure 14 – MCR Modeled Runtimes With Respect to Processors Allocated and Class Size

C – ALC

This section contains graphical analysis of the data collected from ALC.

1 –Runtimes With Respect to Class

Note the point in Figure 15 where the application begins to show non-optimal performance. This nadir, at 16 processors for class S, 64 for class W, and 128 for class A, indicates the processor allocation beyond which TTS degrades rather than improves with the addition of more computational nodes.

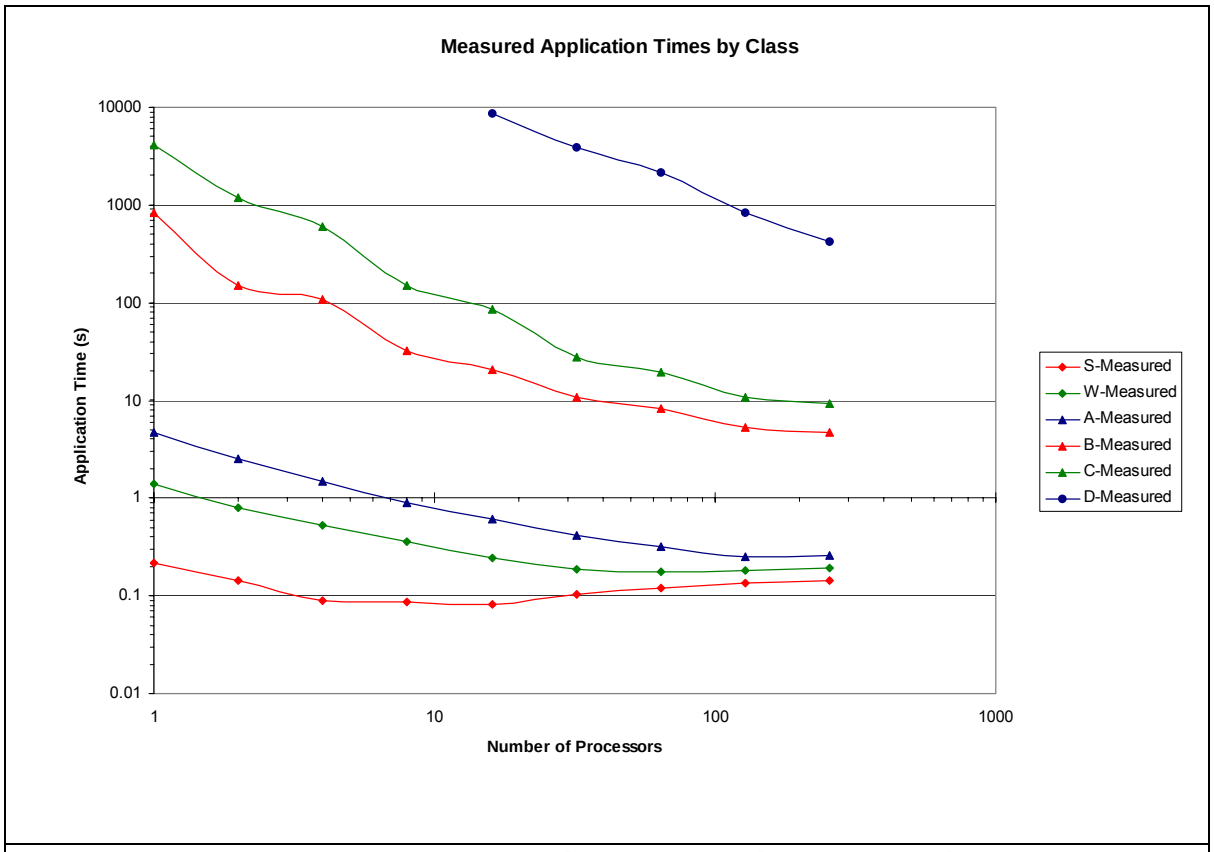

Figure 15 – ALC Measured Runtimes With Respect to Class

Note that Figure 16 is very similar to the plot in Figure 15.

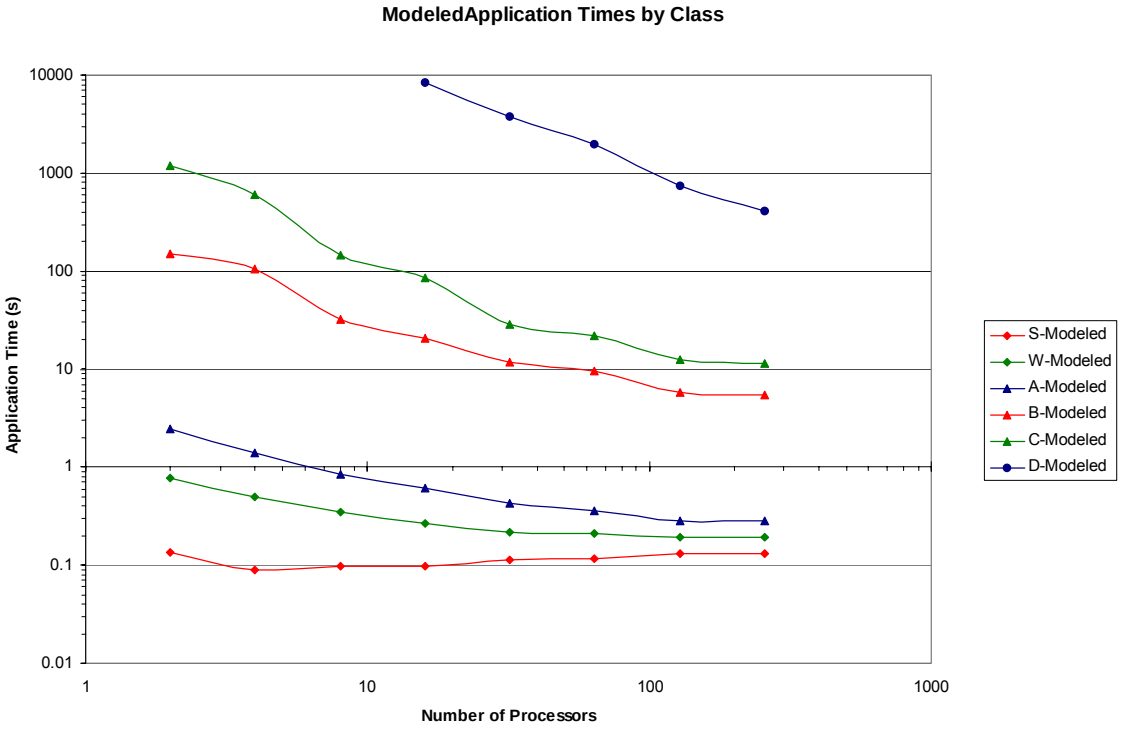

Figure 16 – ALC Modeled Runtimes With Respect to Class

2 –Runtimes with Respect to Processors Allocated

Note the point in Figure 17 below class W for 32 or more processors, indicating the processor allocation is in a region where TTS flattens out rather than improves, as the problem gets smaller.

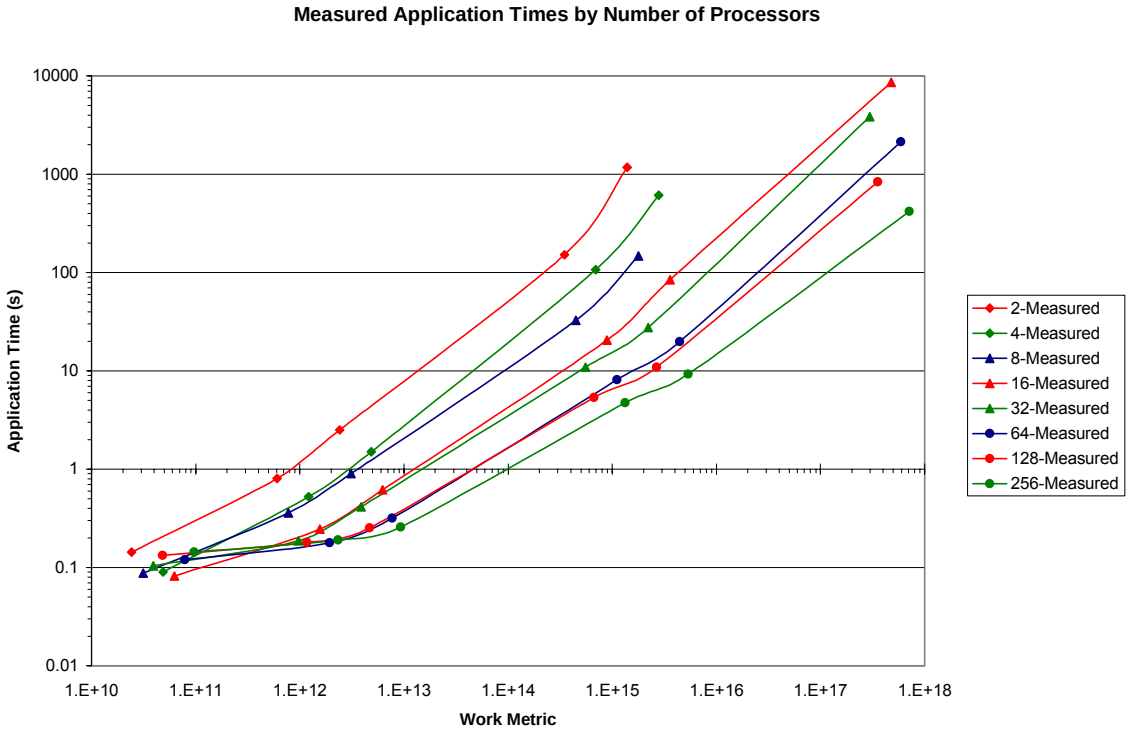

Figure 17 – ALC Measured Runtimes With Respect to Processors Allocated

Note Figure 18 is very similar to the plot in Figure 17.

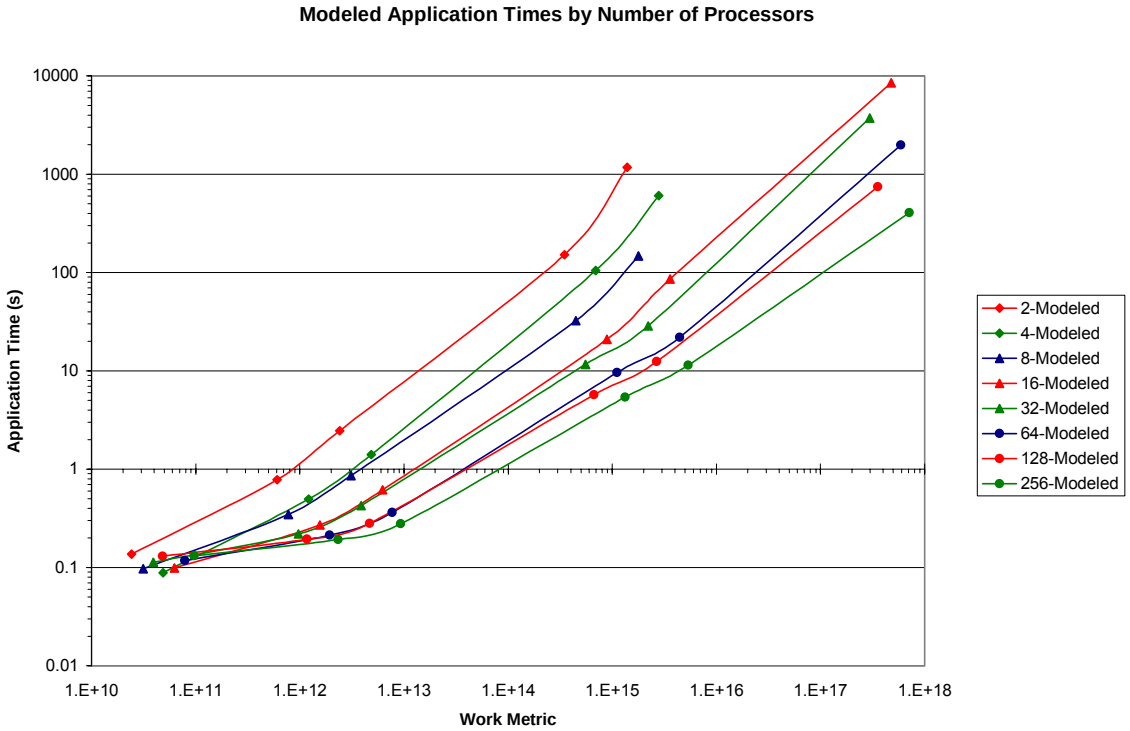

Figure 18 – ALC Modeled Runtimes With Respect to Processors Allocated

3 – Runtimes with Respect to Processors Allocated and Class Size

These graphs are a combination of the above four graphs, and shows the above non-optimal application behavior as valleys along the various contours.

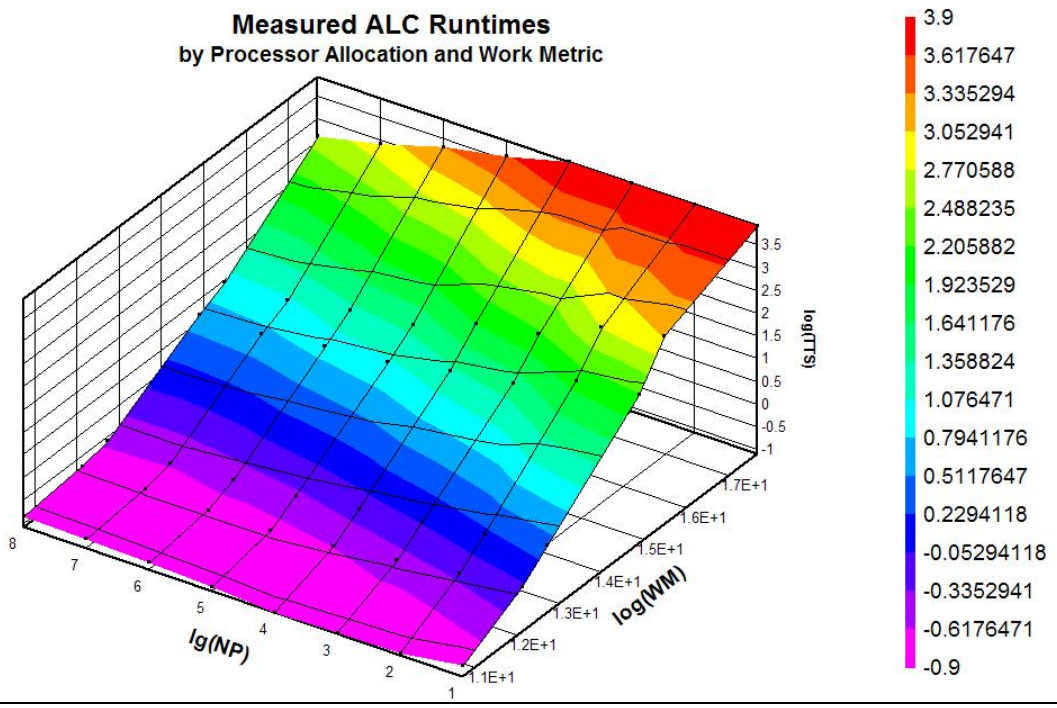

Figure 19 – ALC Measured Runtimes With Respect to Processors Allocated and Class Size

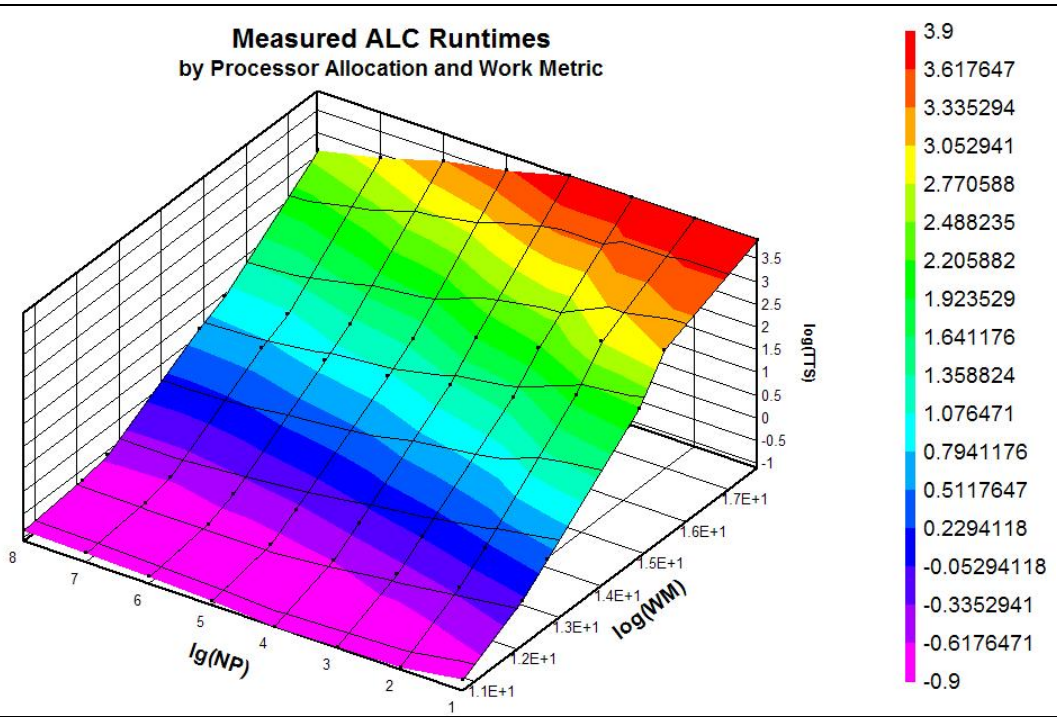

Figure 20 – ALC Modeled Runtimes With Respect to Processors Allocated and Class Size

D – Keck Cluster

This section contains graphical analysis of the data collected from the Keck Cluster.

1 – Runtimes With Respect to Class

Note the point in Figure 21 where the application begins to show non-optimal performance. This nadir, at 16 processors for class S, indicates the processor allocation beyond which TTS degrades rather than improves with the addition of more computational nodes.

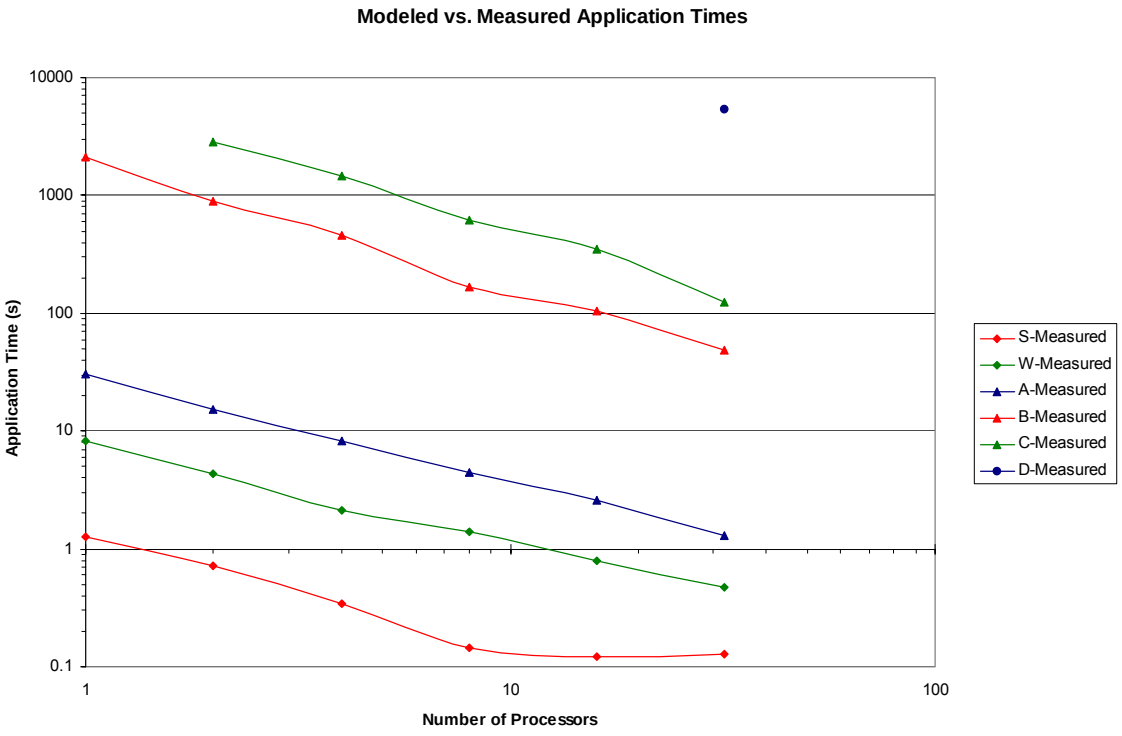

Figure 21 – Keck Cluster Measured Runtimes With Respect to Class

Note that Figure 22 is very similar to the plot in Figure 21.

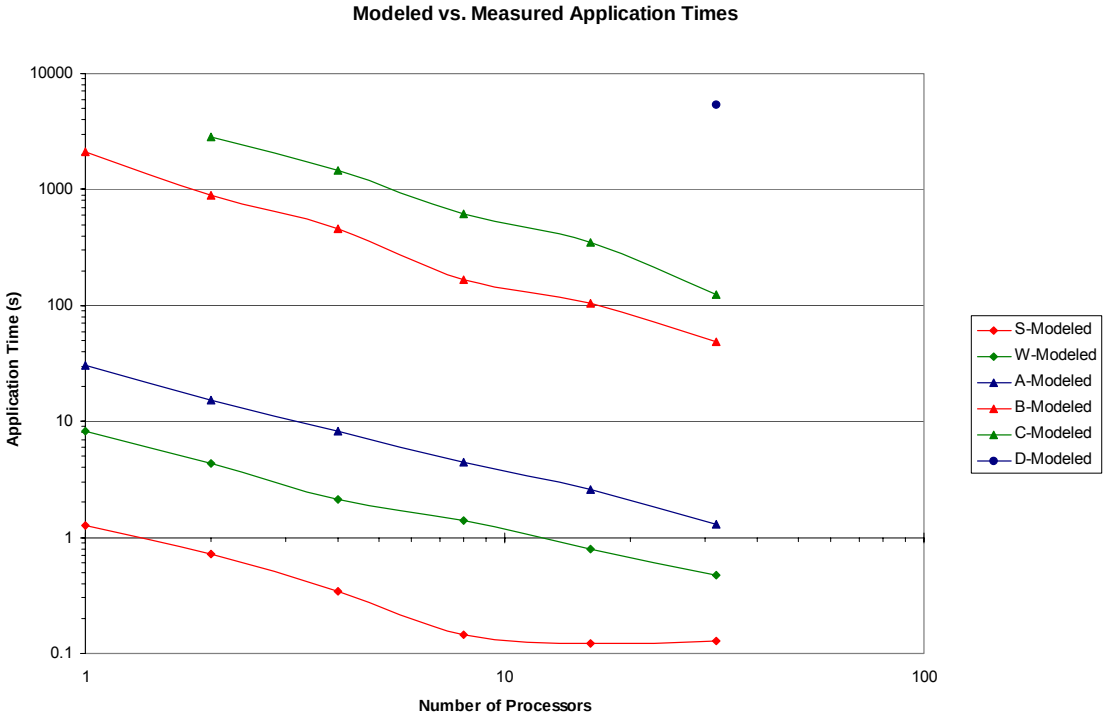

Figure 22 – Keck Cluster Modeled Runtimes With Respect to Class

2 – Runtimes with Respect to Processors Allocated

Note the point in Figure 23 at class S for 32 processors, indicating the processor allocation is in a region where TTS begins to flatten, as the problem gets smaller.

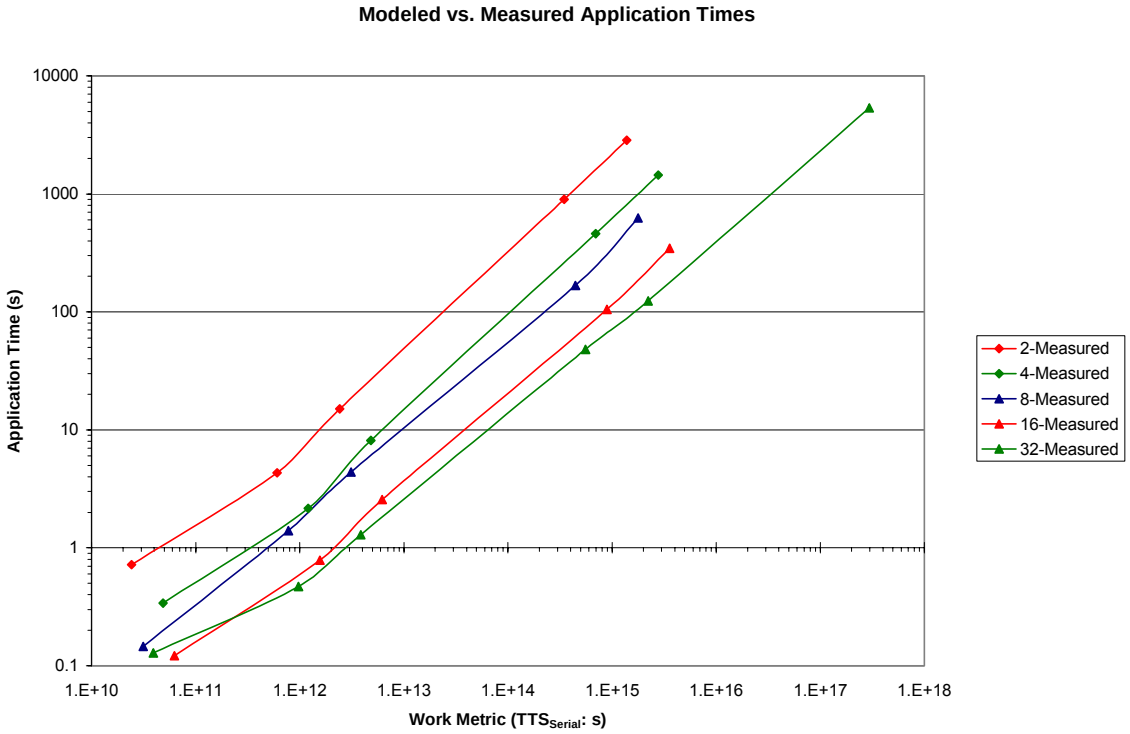

Figure 23 – Keck Cluster Measured Runtimes With Respect to Processors Allocated

Note that Figure 24 is very similar to the plot in Figure 23.

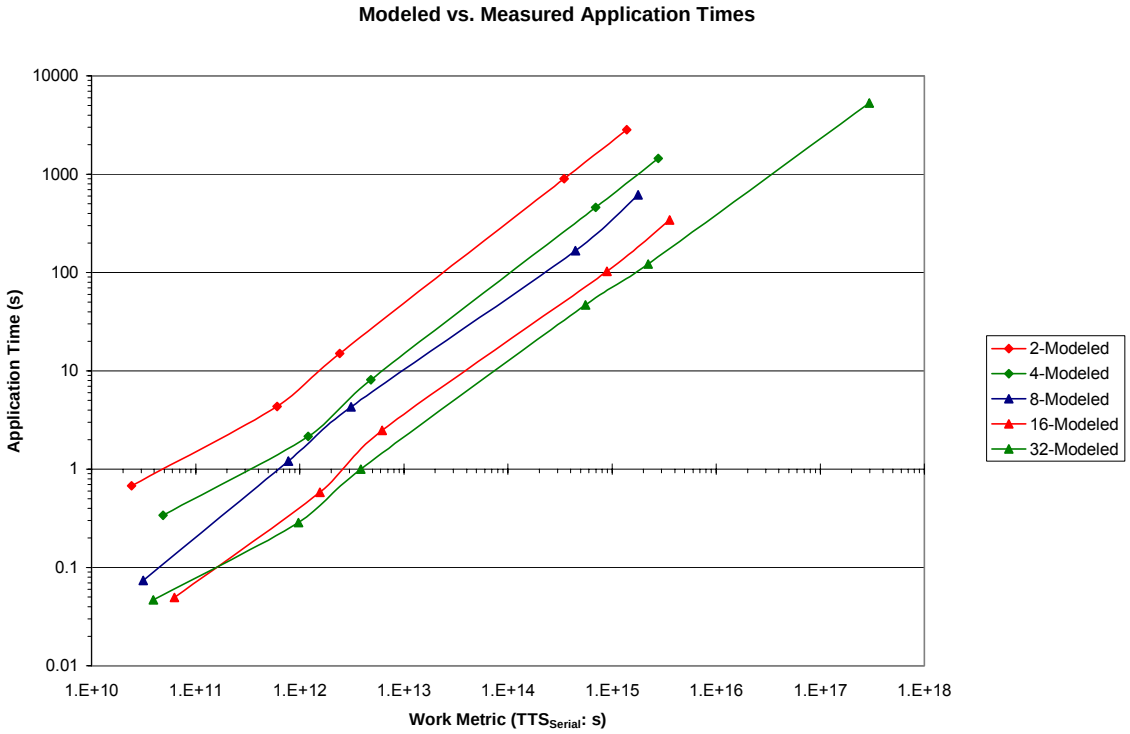

Figure 24 – Keck Cluster Modeled Runtimes With Respect to Processors Allocated

3 – Runtimes with Respect to Processors Allocated and Class Size

These graphs are a combination of the above four graphs, and shows the above non-optimal application behavior as valleys along the various contours.

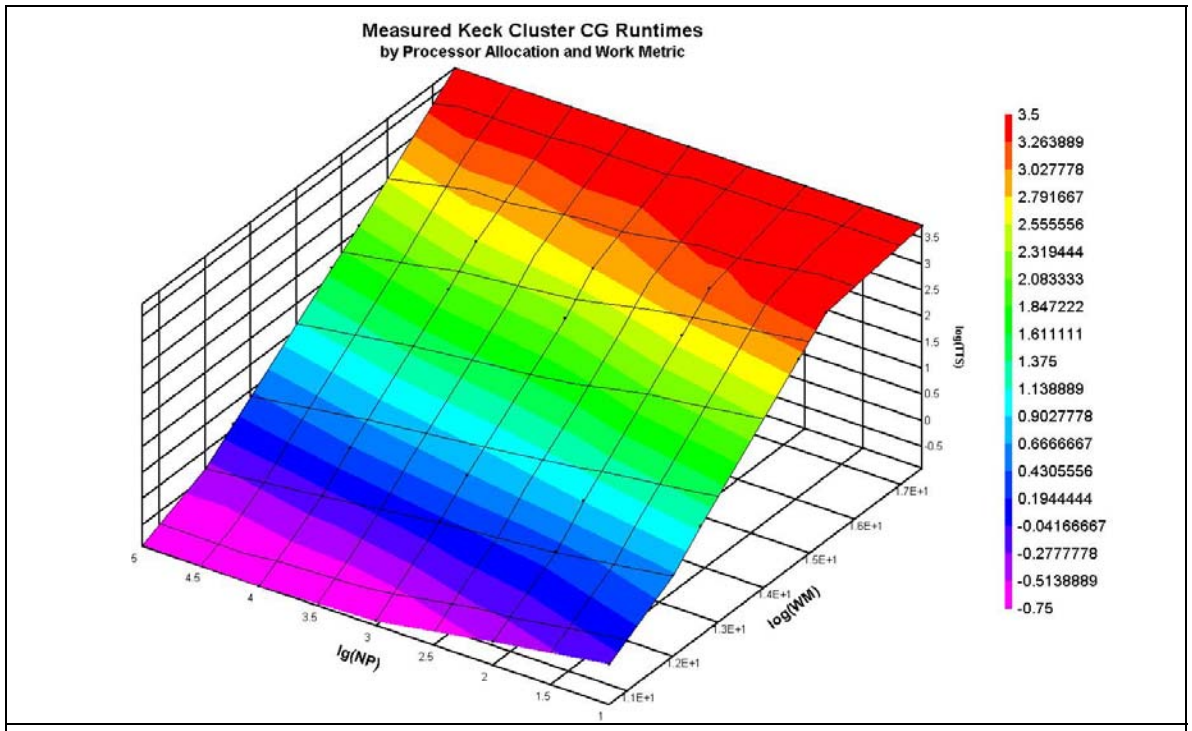

Figure 25 – Keck Cluster Measured Runtimes With Respect to Processors Allocated and Class Size

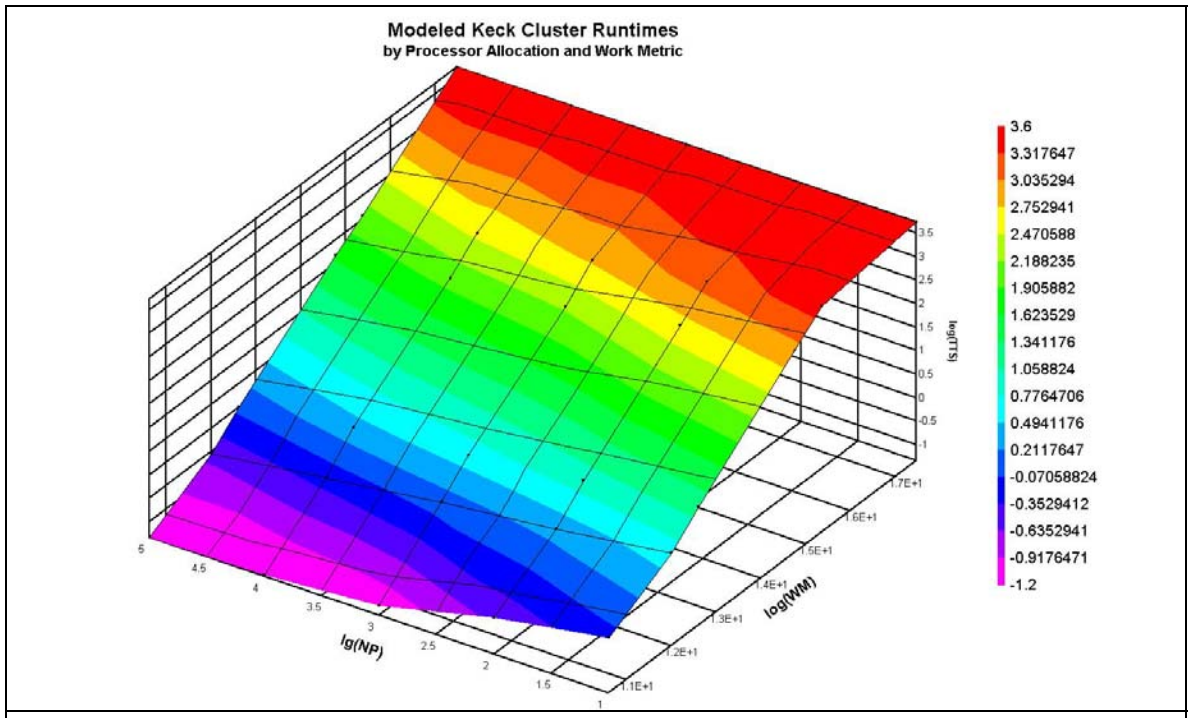

Figure 26 – Keck Cluster Modeled Runtimes With Respect to Processors Allocated and Class Size

E – Analysis of QNM Results as Compared to Measured Results

1 – Relative Error

Presented in this section are the relative errors of the QNM model as compared to the measured system performance. The relative error was determined using Equation 4 below.

$$\%RelativeError = \frac{WCT_{QNM} - WCT_{observed}}{WCT_{observed}}$$

Equation 4 – Determining Relative Error in QNM Models

1.1 – Relative Error on MCR

Figure 27 and Table 6 below show the relative error of the QNM model for MCR. Values range from -50.77% to 24.23% with an average error of 2.41%. All but two of the error calculations fall within the 10 – 30% accuracy range that we typically expect from QNM models. The two anomalous predictions occur when we run very small problems over very large processor allocations, which are atypical of normal MPI programming, and thus are situations we are unlikely to encounter during normal use.

Note that the overhead numbers in Table 6 vary greatly in both directions, even within the same class. Additionally, these values are not monotonic, as might be expected. We speculate that overlap between computation and communication decreases the measured run times, as calculated by our methods. Our QNM technique does not consider this overlap, producing a higher prediction. These higher predictions are in areas where the

amount of communication and computation overlap is almost equal; therefore, overlap is more of an issue. This will be addressed in future research.

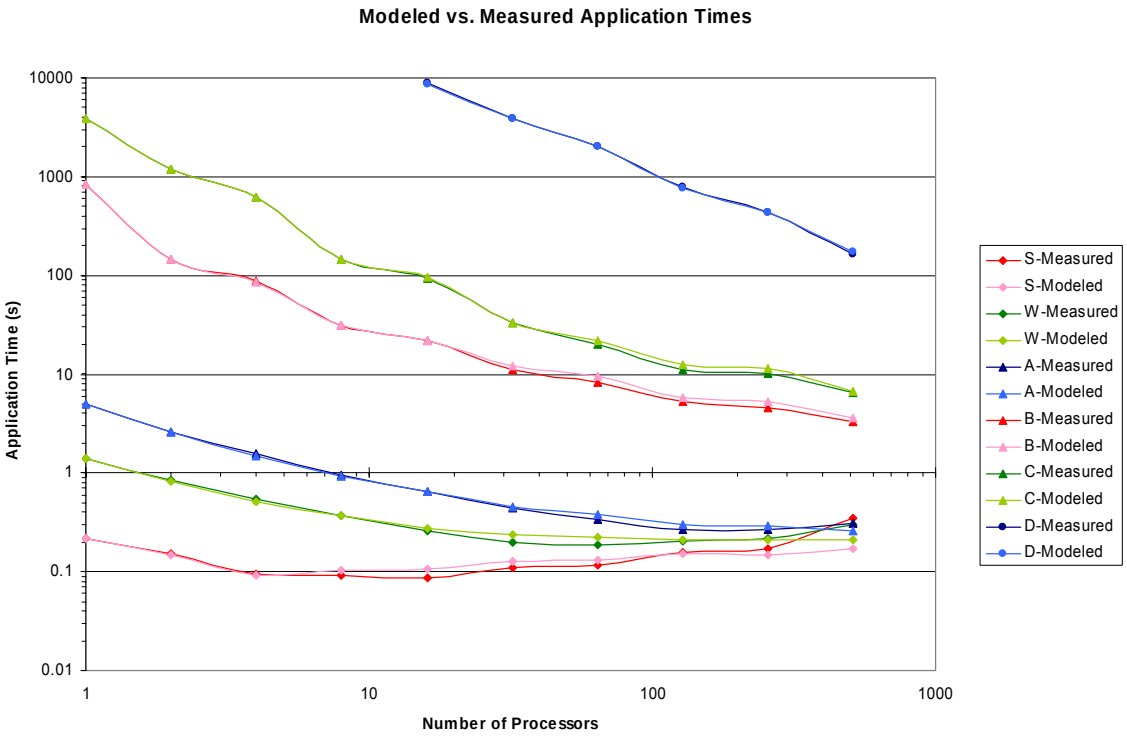

Figure 27 – MCR Measured and Modeled Runtimes With Respect to Class Size

Class		2	4	8	16	32	64	128	256	512
S	54.284E+9	-3.00%	-2.99%	15.24%	24.23%	15.09%	12.36%	-3.86%	-14.03%	-50.77%
W	1.329E+12	-3.14%	-6.44%	-1.57%	7.68%	16.67%	21.12%	5.40%	-3.08%	-30.57%
A	5.301E+12	-1.53%	-5.55%	-3.93%	-0.62%	2.42%	13.10%	11.56%	11.56%	-15.39%
B	759.149E+12	-0.14%	-3.50%	-1.10%	-0.91%	6.36%	15.96%	8.56%	16.49%	10.06%
C	3.036E+15	-0.02%	-1.04%	1.25%	1.43%	1.96%	10.77%	14.44%	11.90%	4.54%
D	474.667E+15				-1.85%	-0.63%	0.52%	1.20%	3.32%	13.30%

Table 6 – Relative Error of QNM on MCR

1.2 – Relative Error on ALC

Figure 28 and Table 7 below shows the relative error of the QNM model for ALC. Values range from -10.86% to 22.50% with an average error of 3.61%. All of the error calculations fall within the 10 – 30% accuracy range that is typically expected from QNM models.

Note that the overhead numbers in Table 7 vary greatly in both directions, even within the same class, as with MCR. Additionally, these values are not monotonic, as might be expected.

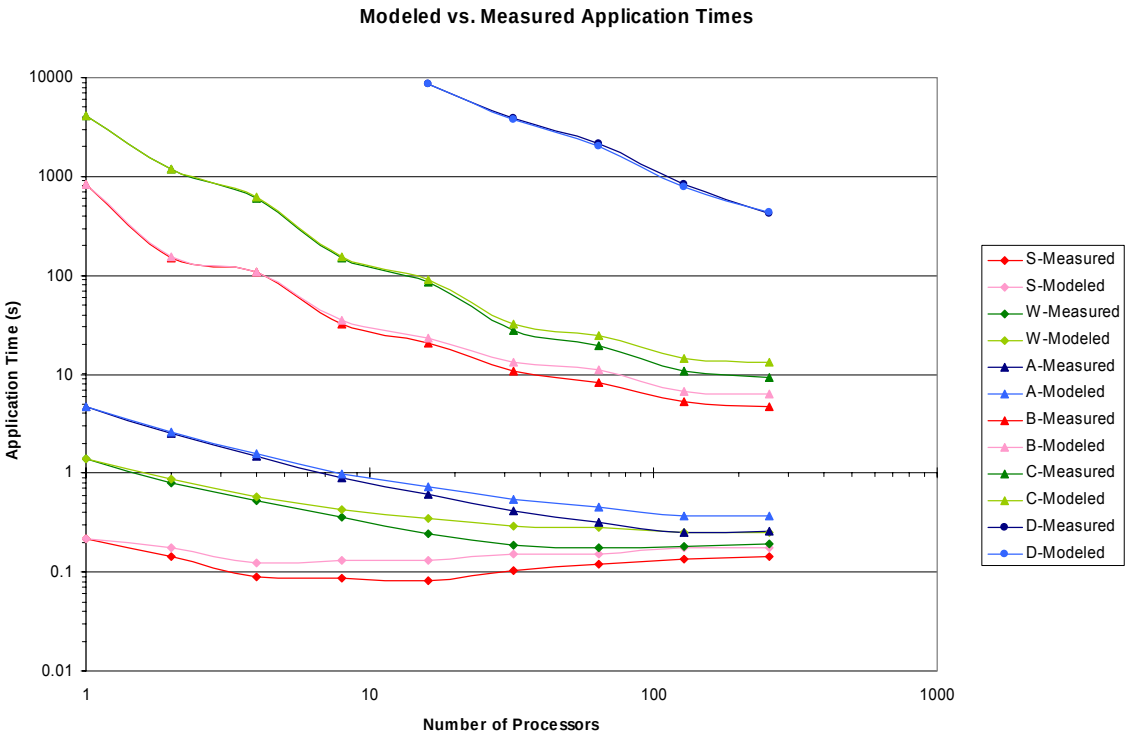

Figure 28 – ALC Measured and Modeled Runtimes With Respect to Class Size

Class		2	4	8	16	32	64	128	256
S	53.724E+9	-4.34%	-1.25%	11.79%	20.43%	9.60%	-2.17%	-2.06%	-8.30%
W	1.322E+12	-2.80%	-6.14%	-3.51%	10.15%	16.91%	19.51%	7.83%	1.34%
A	5.278E+12	-1.67%	-5.75%	-4.99%	-0.23%	2.59%	13.79%	10.58%	7.80%
B	756.356E+12	-0.29%	-2.29%	-0.78%	1.39%	6.91%	18.14%	7.33%	15.00%
C	3.025E+15	-0.01%	-0.87%	-0.42%	1.38%	4.21%	10.99%	14.85%	22.50%
D	486.357E+15				-1.16%	-2.54%	-7.07%	-10.86%	-2.89%

Table 7 – Relative Error of QNM on ALC

1.3 – Relative Error on the Keck Cluster

Figure 29 and Table 8 below shows the relative error of the QNM model for the Keck Cluster. Error values range from -63.76% to -0.02% with an average error of -10.73%. All but four of the error calculations fall within the 10 – 30% accuracy range that we typically expect from QNM models. The four anomalous predictions, again, occur when we run very small problems over larger processor allocations, which is atypical for MPI usage. The Keck Cluster is a much smaller machine than either MCR or ALC, resulting in longer computation times and slower network data transfers.

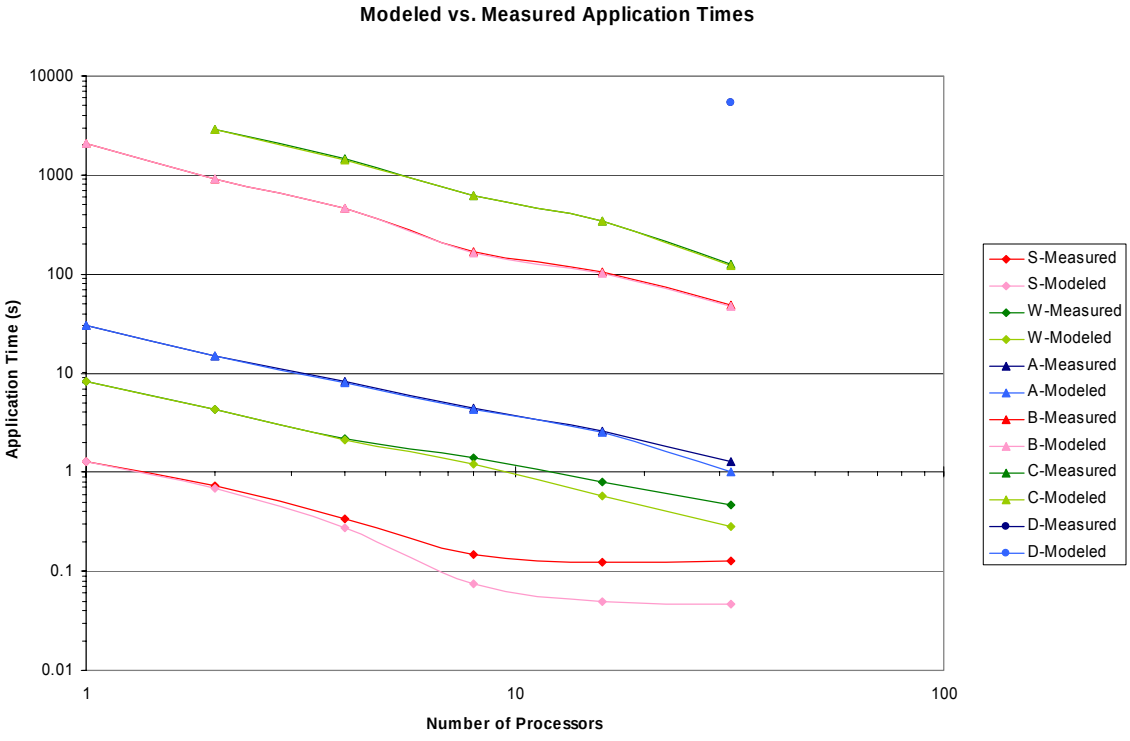

Figure 29 – Keck Cluster Measured and Modeled Runtimes With Respect to Class Size

Class		2	4	8	16	32
S	41.251E+9	-5.39%	-18.92%	-49.26%	-59.41%	-63.76%
W	1.025E+12	-0.23%	-2.50%	-13.76%	-25.87%	-39.35%
A	4.095E+12	-0.20%	-1.06%	-2.30%	-3.32%	-22.24%
B	587.498E+12	-0.08%	-0.60%	-0.70%	-2.07%	-2.63%
C	2.35E+15	-0.02%	-1.04%	-0.89%	-1.32%	-1.84%
D	296.097E+15	-0.63%	-0.63%	-0.63%	-0.63%	-0.63%

Table 8 – Relative Error of QNM on the Keck Cluster

2 - Summary

QNM, indeed, does provide a reasonable means of determining times to solution for the various experimental systems. Only six of the modeled values fell out of the 10 – 30% typical accuracy range of QNM, all of which were under atypical run conditions. The average relative errors of 2.41% on MCR, 3.61% on ALC, and -10.73% on the Keck Cluster are all typical of the results expected from QNM, and show that QNM is a useful tool for determining the TTS for large-scale problems on clustered high-performance computers.

V – Problems Encountered and Further Research

A – Regionalization and Trending

Many components of cluster computers exhibit piecewise linear behavior. For example, both the performance of the system switching mechanism and the caching hierarchy exhibit this type of behavior. For each of these, a plot of the performance can be divided into regions at the inflection points of the graph, and within each region, a particular linear trend dominates. This trend provides reasonably accurate interpolation for intermediate values. This section explores initial attempts to examine if clusters exhibit this piecewise linear behavior.

1 – Baseline Analysis and Results

Examination of the graphs in Chapter IV, Section B (for instance, Figure 9) above indicates that the application exhibits different behavior at certain critical processor allocations for each class. We note this differing behavior by the U-shaped structure of the runtimes for small classes. Typically, we expect the application to produce shorter and shorter runtimes as the number of processors allocated to solving a problem increases. However, beyond the critical point, the application actually requires more and more time to derive a solution, which is counter-intuitive. To account for this counter-intuitive behavior we explored a regime structure to determine what metrics were usable to predict the movement from the intuitive, typical application performance region to the counter-intuitive, undesired application performance region. We used trendlining over small subsets of the data to attempt prediction of these critical processor allocations that

mark the boundary between regions. Results indicate this could be a useful metric for determining where these regions lie and that further investigation is required.[52]

2 – Percent CPU Utilization as Regime Change Metric

Initial analysis of the measured results shows that percent CPU utilization may be a good indicator for movement from the typical to the atypical regions of application behavior. As more processors are allocated to a problem, the percentage of the CPU time used to solve that problem steadily decreases, as more system time is needed for network traffic. When this utilization drops below 70 – 80 %, the application moves into the atypical behavior region, and begins to take longer to derive solutions than smaller processor allocations for the same problem. To predict where this nadir occurs without having to run the program, we applied predictive linear trendlines for one, two, four, and eight processors to the data and extrapolated until the trendlines crossed the critical barrier of 80% (MCR) and 70% (ALC), where the application begins the atypical behavior described above. The resulting trendline equation, when solved for the number of processors at the critical CPU utilization percentage and rounded up to the next power of two, provides a good estimate of the number of processors at the critical allocation point. We analyzed classes S, W, and A, as these were the only classes exhibit such atypical application behavior in our runs.

[52] Due to the nature of the system, CPU utilization data was not able to be collected for the Keck Cluster, and will not be discussed in this section.

2.1 – MCR Percent CPU Utilization and Trends

The following graphs and tables show the results of applying this trending analysis to MPI runtimes on MCR.

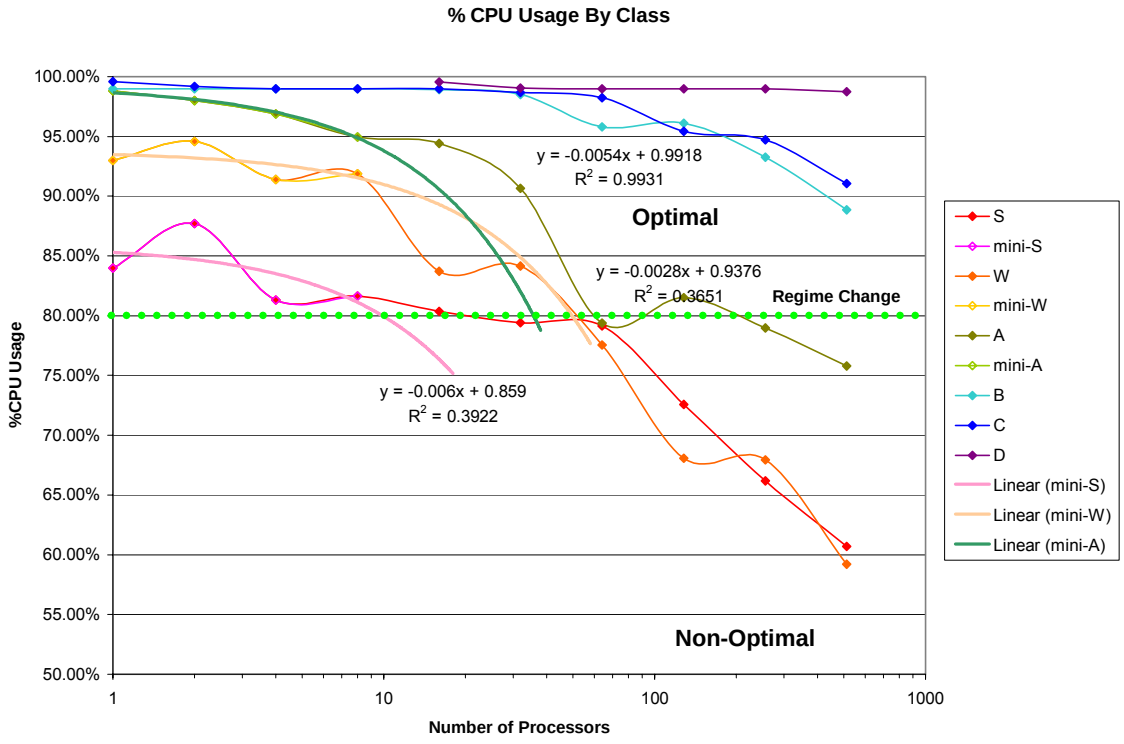

Figure 30 – MCR Percent CPU Utilization and Trendlines

Class	Trendline Equation	Predicted Critical CPU Allocation	Predicted Critical CPU Allocation (Rounded Up)	Actual Measured Critical CPU Allocation
S	$y = -0.006x + 0.859$	10	16	16
W	$y = -0.0028x + 0.9376$	49	64	64
A	$y = -0.0054x + 0.9918$	36	64	256

Table 9 – MCR Trendline Equations, Predictions, and Measured Results

For classes S and W, the predictive trendline accurately predicts the critical CPU allocation beyond which the application exhibits atypical behavior. For class A, the result from the trendline equation is within two processor allocations of the critical

allocation, indicates a leveling-off of the runtimes, and shows the entrance into atypical application behavior area.

2.2 – ALC Percent CPU Utilization and Trends

The following graphs and tables show the results of applying this trending analysis to MPI runtimes on ALC.

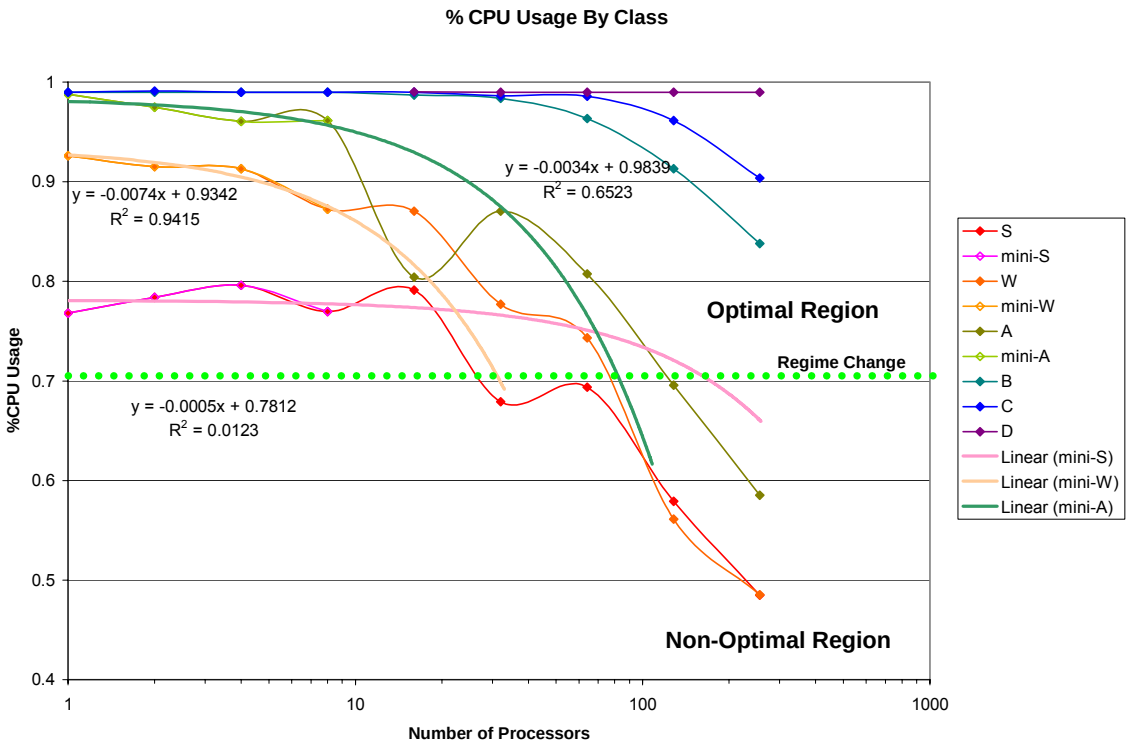

Figure 31 – ALC Percent CPU Utilization and Trendlines

Class	Trendline Equation	Predicted Critical CPU Allocation	Predicted Critical CPU Allocation (Rounded Up)	Actual Measured Critical CPU Allocation
S	$y = -0.0005x + 0.7812$	168	256	16
W	$y = -0.0074x + 0.9342$	32	32	64
A	$y = -0.0034x + 0.9839$	84	128	128

Table 10 – ALC Trendline Equations, Predictions, and Measured Results

For class A the predictive trendline accurately predicts the critical CPU allocation beyond which the application exhibits atypical behavior. For class W, the result from the trendline equation is within one processor allocation of the critical allocation, indicates a leveling-off of the runtimes, and shows the entrance into the atypical application behavior area. Class S, which exhibits nearly constant behavior on this system for very small processor allocations, grossly over-predicts the CPU allocation for the critical region. This suggests that while percentage CPU utilization may be a useful tool in predicting entrance into the critical region, other tools are necessary to verify the results, particularly if the application is exhibiting near constant behavior.

B – Model Input Parameterization and Trending

1 – Initial Analysis and Results

This thesis has focused on the viability of QNM to represent accurately system behavior based on collected results from actual program execution. However, QNM would be much more useful if it were able to predict system performance without having to perform actual program runs and analyze the collected performance data. One possible means of doing this would be to execute small versions of the problem over small processor allocations and estimate the values for combinations of larger versions of the problem and larger processor allocations. We examined several methods of deriving these estimates, mostly with disappointing results. However, one method, predictive trendlining of the components of execution time, shows promise, and is the focus of this section.

To understand better the program execution times, they were broken down into various components of execution time, using data gathered by mpiP or calculated by the model. In particular, three components were highly important: MPI Active Time, MPI Wait Time, and Compute Time. MPI Wait Time is the amount of time the computer spends inside MPI calls, but not working actively on that call (i.e. blocked for I/O). This was determined by taking the MPI Wait readings directly from the mpiP output. MPI Active Time is the amount of time the computer spends inside MPI calls and actively working on processing those calls (i.e.: executing code in the MPI routines). We determined this by taking the total time reported in MPI calls (MPI_Time) and subtracting the MPI_Wait time. Compute Time is the amount of time the computer spends outside any MPI calls (i.e.: is not within an MPI routine, or blocked for an MPI routine). This was determined by subtracting the total MPI Time from the total Application Time. In the following graphs, the darker solid triangles represent data collected from the experimental systems, while the lighter hollow squares represent data calculated from the QNM model. The experimental computation values equate to the modeled compute time, whereas MPI Active uses the same label for both experimental and modeled values.

2 – MCR Analysis and Results

We plotted the various components of creating a QNM model for MCR, samples of which we show below. For both the Computation and MPI Active components, we fit a power trendline to the data, creating a linear correlation in the logarithmic domain. Strong R^2 values (> 0.9) for the trend of MPI Active time suggest that MPI Active is a strong candidate for prediction using trendlines. Weak R^2 values (< 0.9) for the trend of

Computation time suggest that Computation is not a good candidate for prediction using trendlines. We ruled out trending of MPI Wait on visual inspection.

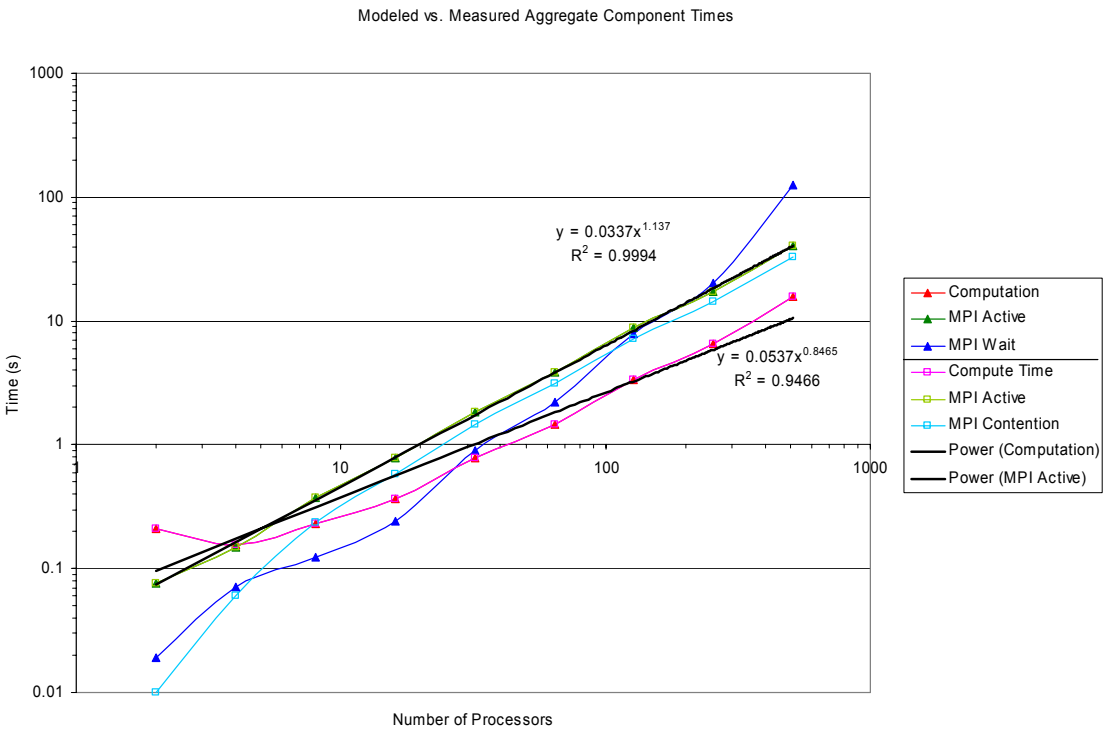

Figure 32 – MCR CG Class S Component Times with Trendline

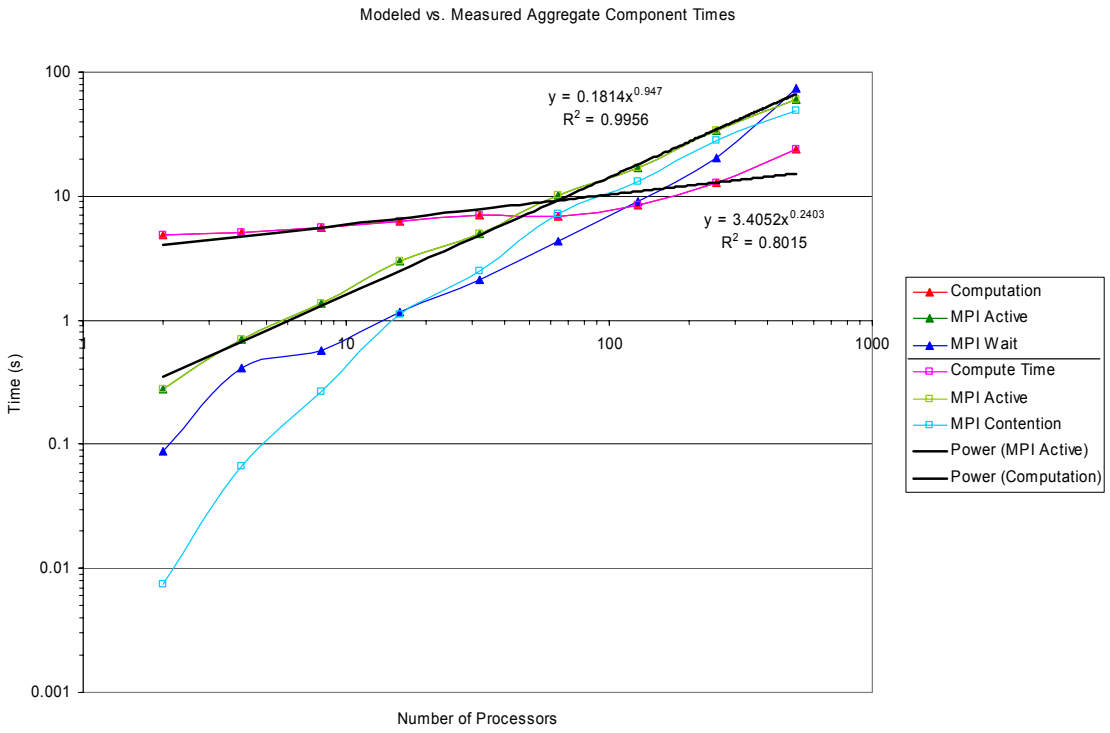

Figure 33 – MCR CG Class A Component Times with Trendline

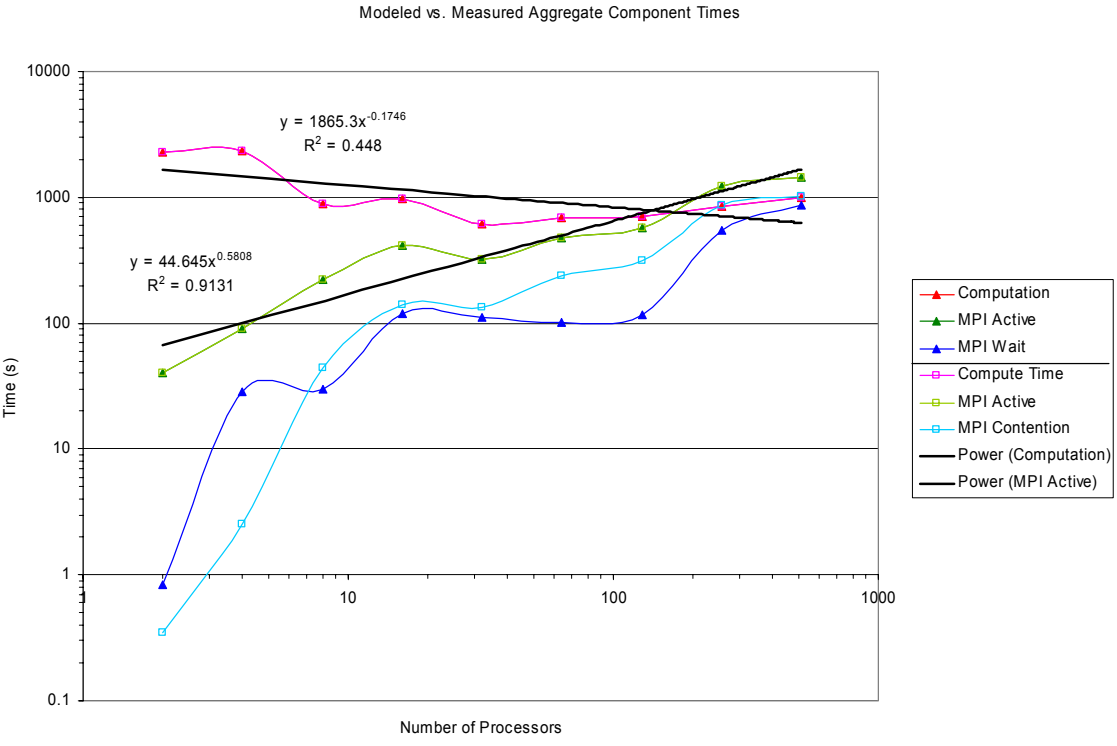

Figure 34 – MCR CG Class C Component Times with Trendline

3 – ALC Analysis and Results

We plotted the various components of creating a QNM model ALC, samples of which we show below. For both the Computation and MPI Active components, we fit a power trendline to the data, creating a linear correlation in the logarithmic domain. Strong R^2 values (> 0.9) for the trend of MPI Active time suggest that MPI Active is a strong candidate for prediction using trendlines. Weak R^2 values (< 0.9) for the trend of Computation time suggest that Computation is not a good candidate for prediction using trendlines. We ruled out trending of MPI Wait was ruled out on visual inspection.

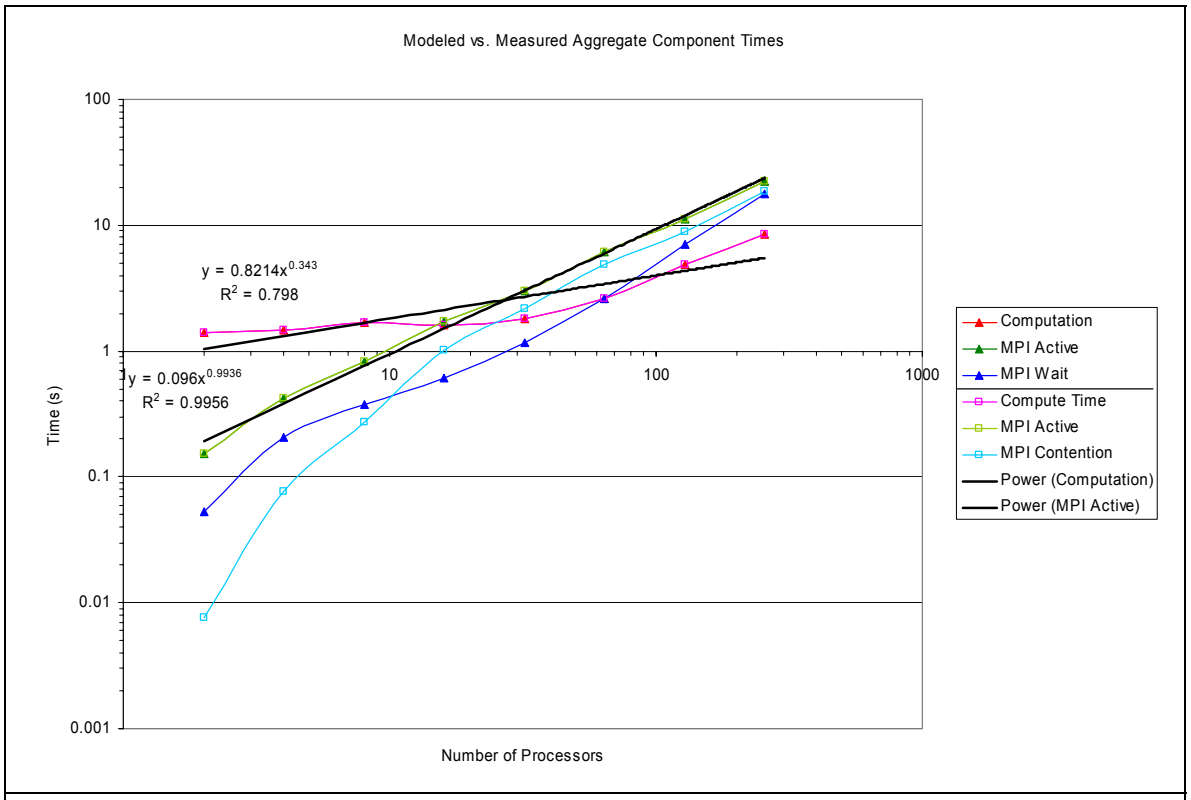

Figure 35 – ALC CG Class W Component Times with Trendline

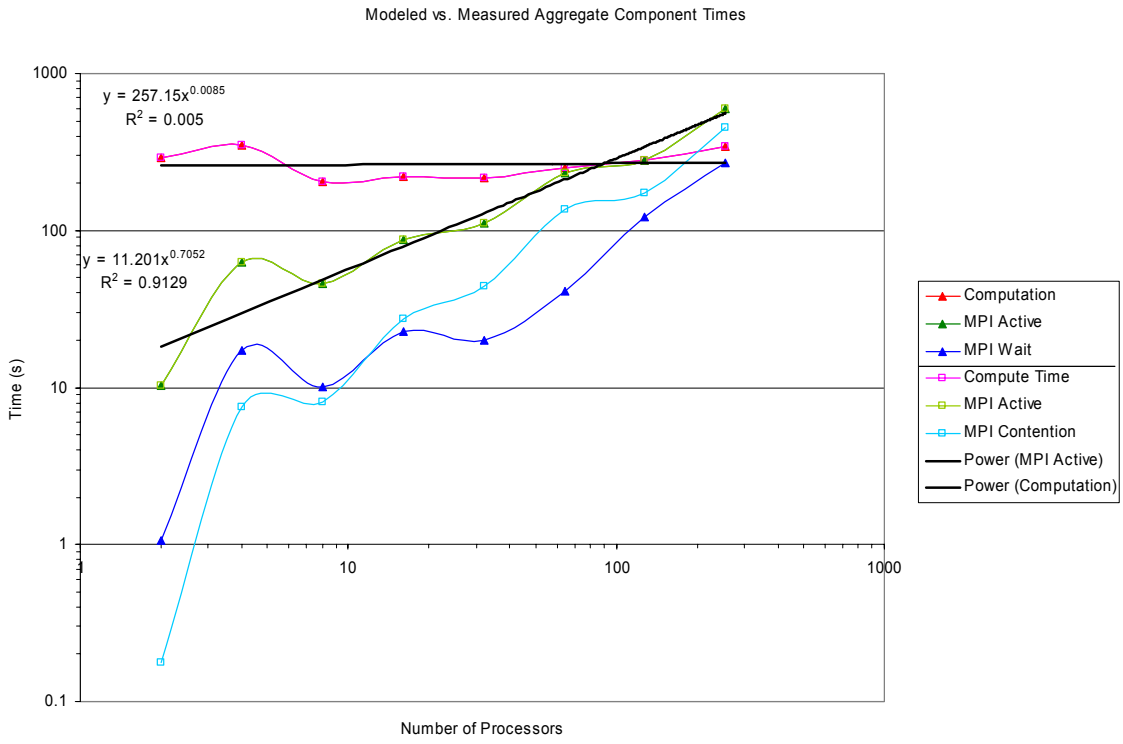

Figure 36 – ALC CG Class B Component Times with Trendline

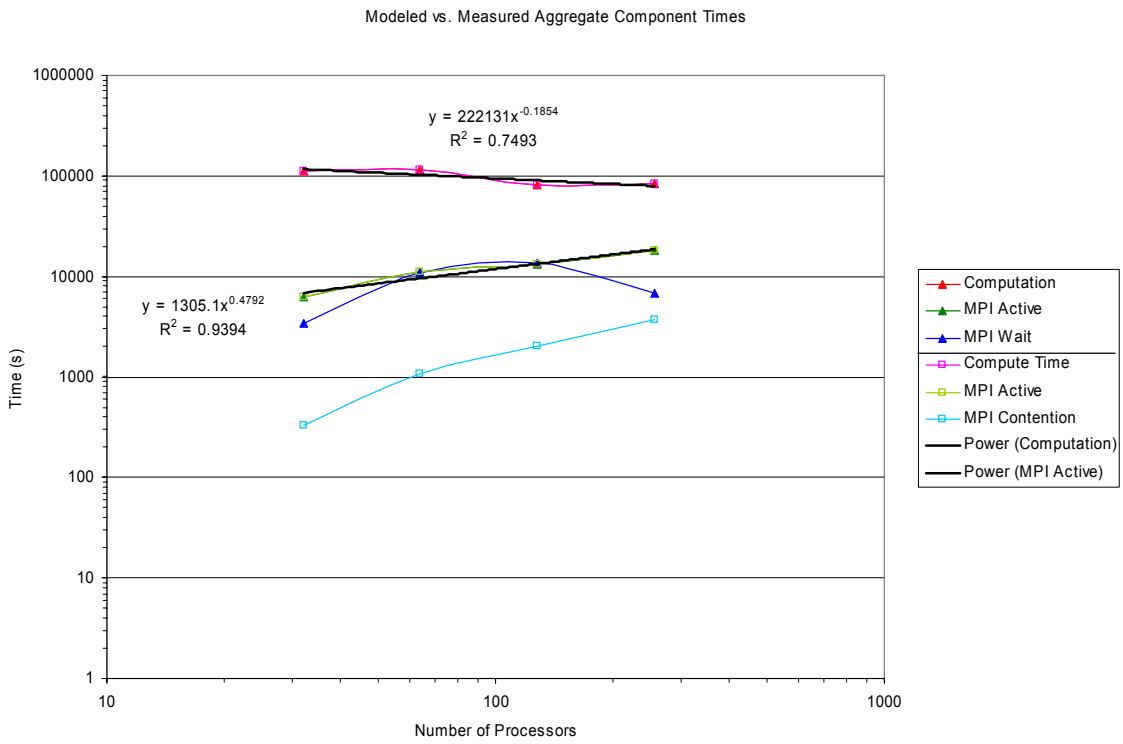

Figure 37 – ALC CG Class D Component Times with Trendline

4 – Keck Cluster Analysis and Results

We plotted the various components of creating a QNM model for the Keck Cluster, samples of which we show below. For both the Computation and MPI Active components, we fit a power trendline to the data, creating a linear correlation in the logarithmic domain. Strong R^2 values (> 0.9) for the trend of MPI Active time suggest that MPI Active is a strong candidate for prediction using trendlines. Weak R^2 values (< 0.9) for the trend of Computation time suggest that Computation is not a good candidate for prediction using trendlines. We ruled out trending of MPI Wait on visual inspection

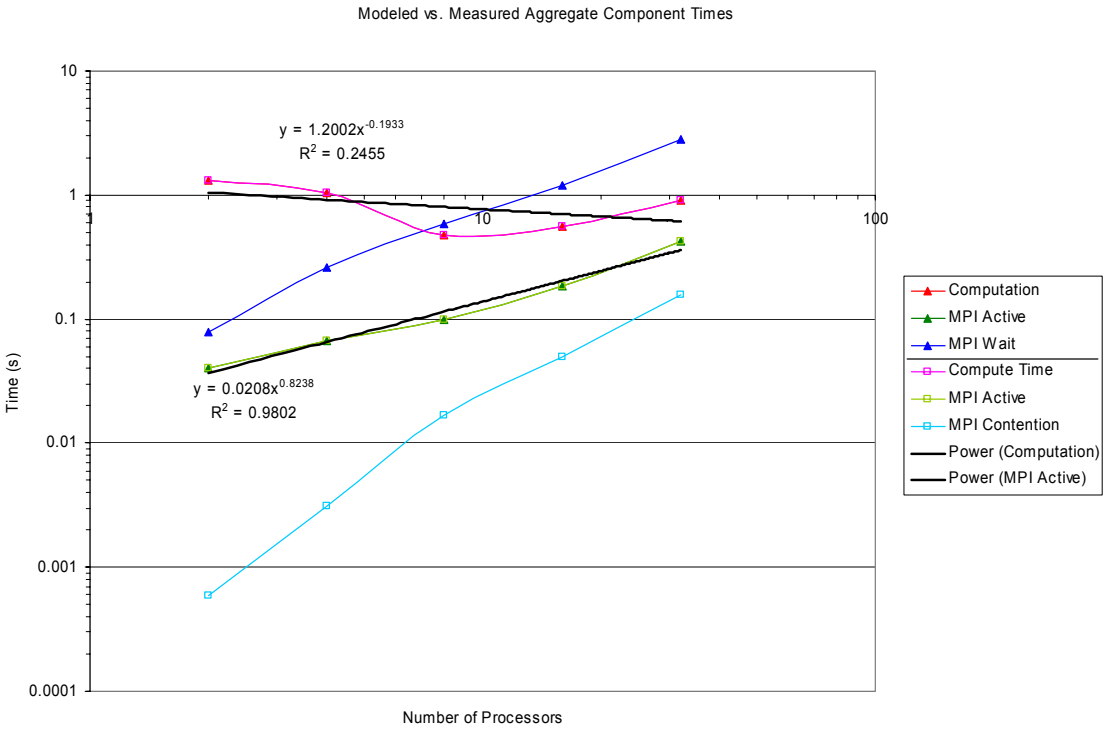

Figure 38 – Keck Cluster CG Class S Component Times with Trendline

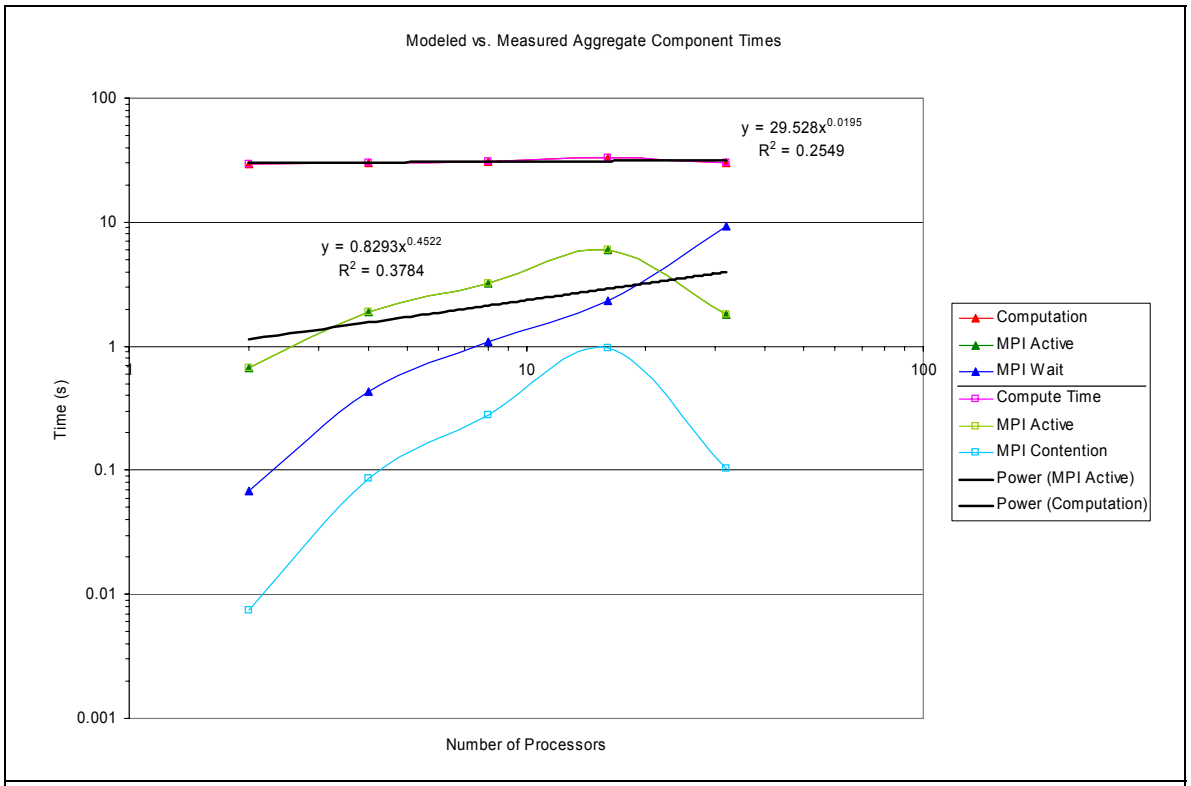

Figure 39 – Keck Cluster CG Class A Component Times with Trendline

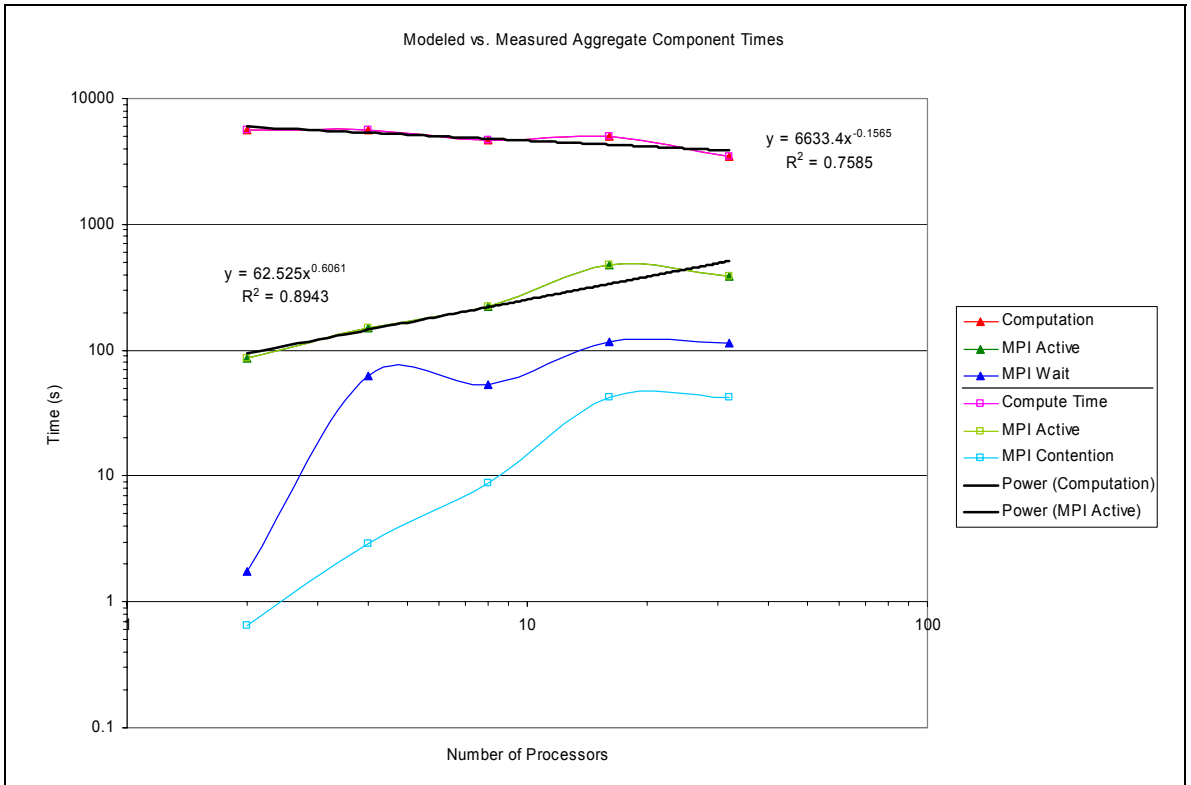

Figure 40 – Keck Cluster CG Class C Component Times with Trendline

5 – Analysis Summary

The tables below give the power trendline equation and R^2 for each of the machines and classes as the problem scales strongly. We plan analysis and trending as the problem scales weakly for the near future.

QNM models MPI Wait as MPI Contention, which we do not consider in the above graphs or results below, since it is a value calculated from MPI Active by the QNM solver. Compute Time, while showing some potential, was less than successful, with 15 of 17 results falling below the desirable R^2 value of 0.9. MPI Active, on the other hand, does show much promise for trend modeling, as all but four of the results fell above the desired R^2 value.

Machine	MCR	ALC	Keck Cluster
Class S	$y = 0.0537x^{0.8465}$ $R^2 = 0.9466$	$y = 0.0612x^{0.7677}$ $R^2 = 0.9318$	$y = 1.2002x^{-0.1933}$ $R^2 = 0.2455$
Class W	$y = 0.7015x^{0.4348}$ $R^2 = 0.8339$	$y = 0.8214x^{0.343}$ $R^2 = 0.798$	$y = 8.2092x^{0.0079}$ $R^2 = 0.0124$
Class A	$y = 3.0452x^{0.2403}$ $R^2 = 0.8015$	$y = 3.749x^{0.1706}$ $R^2 = 0.8623$	$y = 29.528x^{0.0195}$ $R^2 = 0.2549$
Class B	$y = 212.9x^{0.0763}$ $R^2 = 0.3988$	$y = 257.15x^{0.0085}$ $R^2 = 0.005$	$y = 1{,}906.3x^{-0.1177}$ $R^2 = 0.572$
Class C	$y = 1{,}865.3x^{-0.1746}$ $R^2 = 0.448$	$y = 2{,}220.9x^{-0.2559}$ $R^2 = 0.6454$	$y = 6{,}633.4x^{-0.1565}$ $R^2 = 0.7585$
Class D	$y = 390{,}464x^{-0.3171}$ $R^2 = 0.7835$	$y = 222{,}131x^{-0.1854}$ $R^2 = 0.7493$	

Table 11 – Computation Trendline Equations, Predictions, and Measured Results

Machine	MCR	ALC	Keck Cluster
Class S	$y = 0.0337x^{1.137}$ $R^2 = 0.9994$	$y = 0.0313x^{1.1271}$ $R^2 = 0.9995$	$y = 0.0208x^{8236}$ $R^2 = 0.9802$
Class W	$y = 0.0971x^{1.0038}$ $R^2 = 0.9976$	$y = 0.096x^{0.9936}$ $R^2 = 0.9956$	$y = 0.2932x^{0.3144}$ $R^2 = 0.4774$
Class A	$y = 0.1814x^{0.947}$ $R^2 = 0.9956$	$y = 0.1772x^{0.9481}$ $R^2 = 0.9947$	$y = 0.8293x^{0.4522}$ $R^2 = 0.3784$
Class B	$y = 11.842x^{0.6831}$ $R^2 = 0.9732$	$y = 11.201x^{0.7052}$ $R^2 = 0.9129$	$y = 20.185x^{0.6777}$ $R^2 = 0.9826$
Class C	$y = 44.645x^{0.5806}$ $R^2 = 0.9131$	$y = 53.33x^{0.5326}$ $R^2 = 0.9251$	$y = 62.525x^{0.6061}$ $R^2 = 0.8943$
Class D	$y = 2{,}235x^{0.3955}$ $R^2 = 0.8932$	$y = 1{,}305.1x^{0.4792}$ $R^2 = 0.9394$	

Table 12 – MPI Active Trendline Equations, Predictions, and Measured Results

C – Class and Problem Sizes, Work Metric, and Data Set Size

As discussed in Chapter III above, a work metric was necessary to determine the performance of NAS-PB as it relates to the benchmark problem size. Work metrics used in this thesis are highly dependent on collecting data from the program in execution or from detailed analysis of the code, and thus the communication patterns used to solve the problem. While we explored various input parameters for NBP for potential worth as a work metric, no metric or combination of metrics proved satisfactory. This indicates further research, including reexamination of the NPB input parameters, to divorce the work metric from both the need to analyze the program code and the collected observed behavior. Determination of a data set size, which would include the amount of data in memory, which is not dependent on such analysis and data collection is highly desirable and should receive future consideration.

D – Switch Delay versus No Switch Delay

Initial investigation included a delay server, as shown in Figure 1in Chapter I. In later refinement of the model, we found that, for MCR and ALC, we realized better results with the switch delay removed from the model. In practical terms, this means that we recalculated the model with the switch delay set to zero. However, for the Keck Cluster, this was not always the case.

1 – MCR Analysis

Following are sample graphs from MCR showing the measured values (blue), the model excluding the switch delay (magenta), and the model including the switch delay (green).

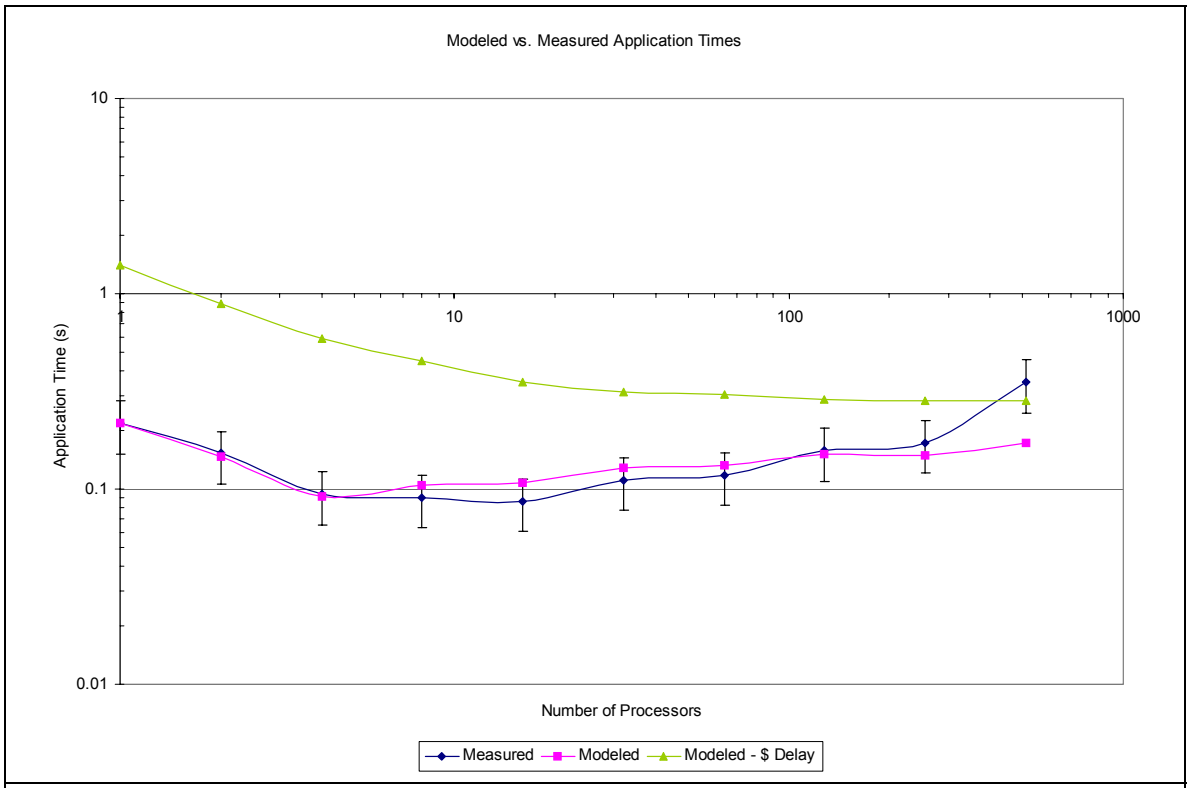

Figure 41 – MCR Class S Model Comparison

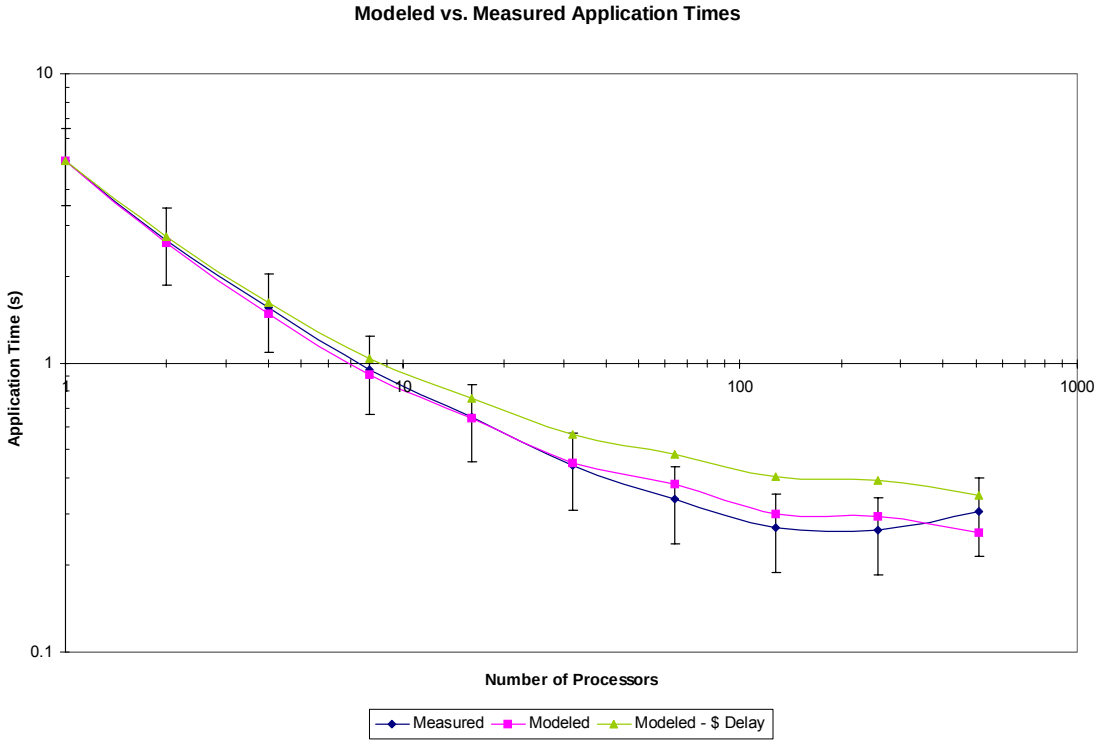

Figure 42 – MCR Class A Model Comparison

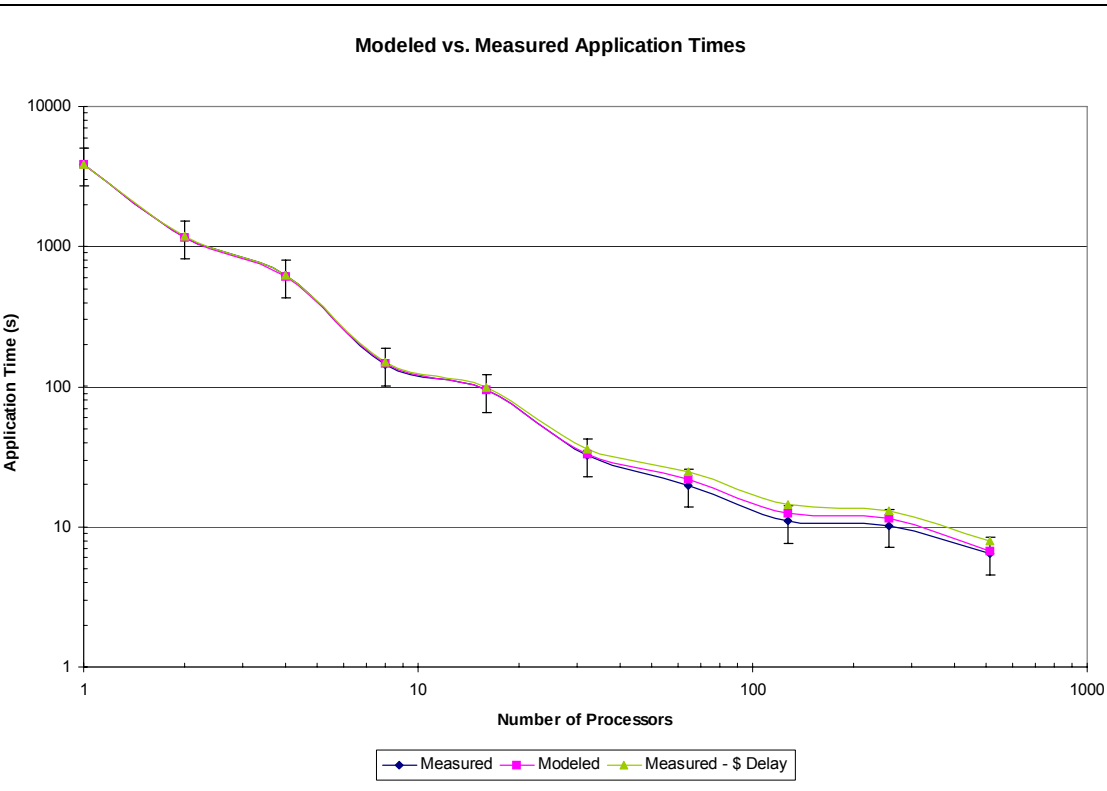

Figure 43 – MCR Class C Model Comparison

2 – ALC Analysis

Following are sample graphs from ALC showing the measured values (blue), the model excluding the switch delay (magenta), and the model including the switch delay (green).

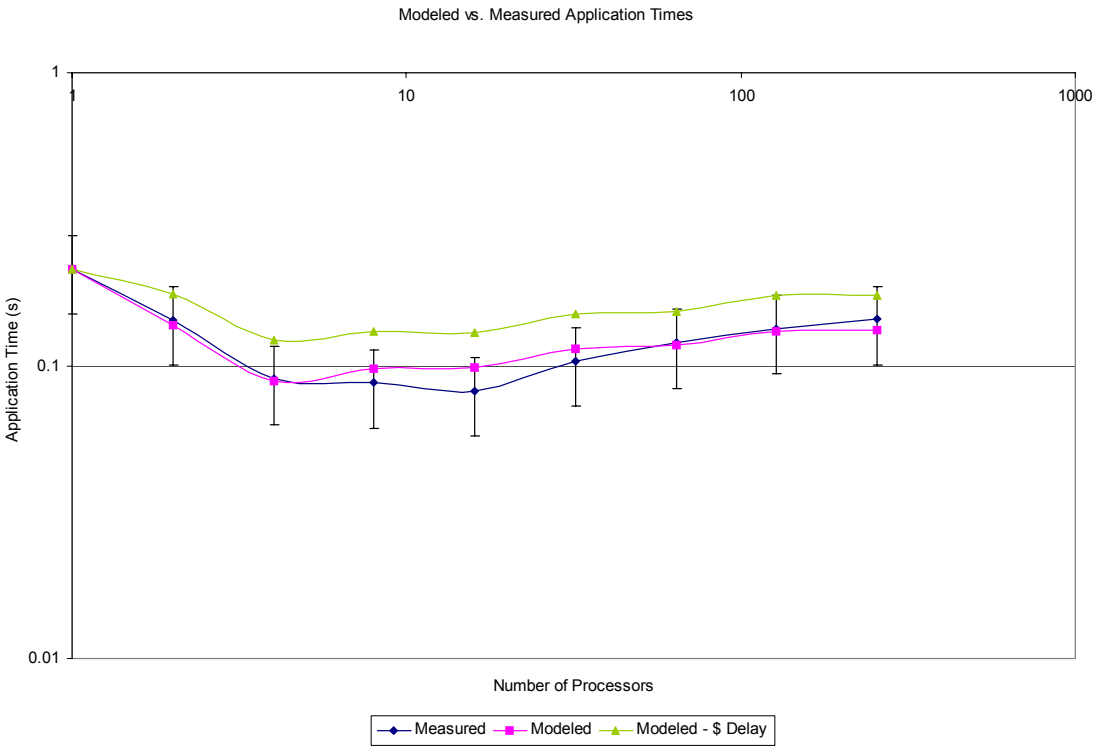

Figure 44 – ALC Class S Model Comparison

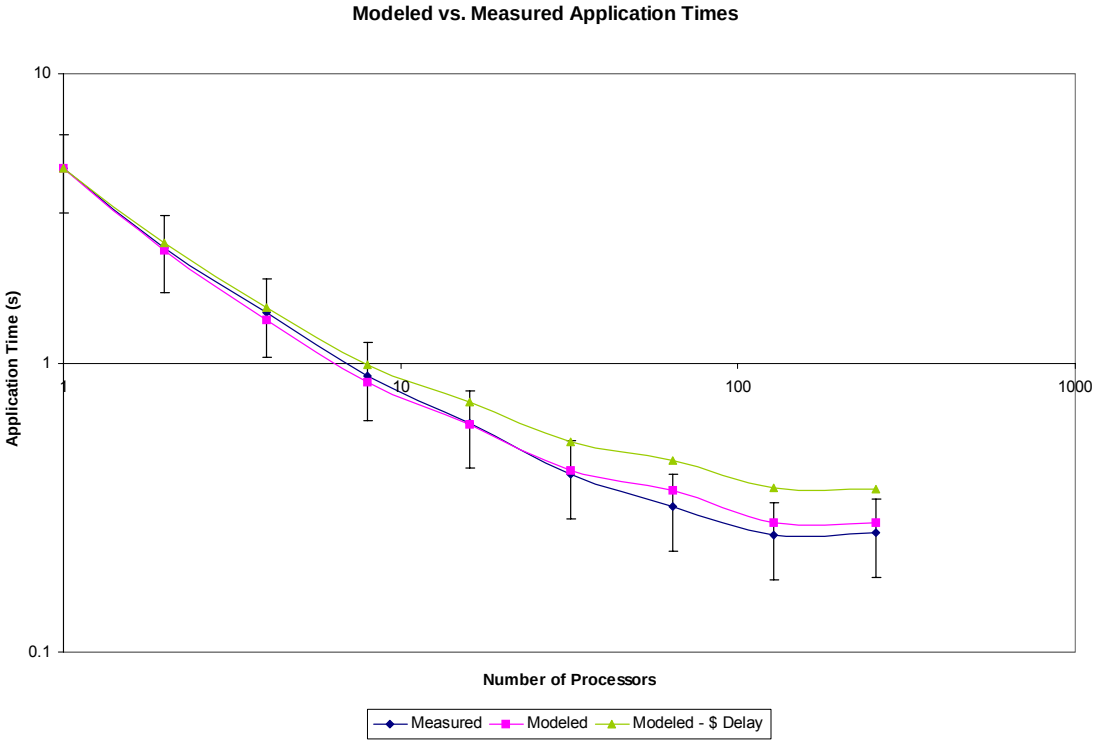

Figure 45 – ALC Class A Model Comparison

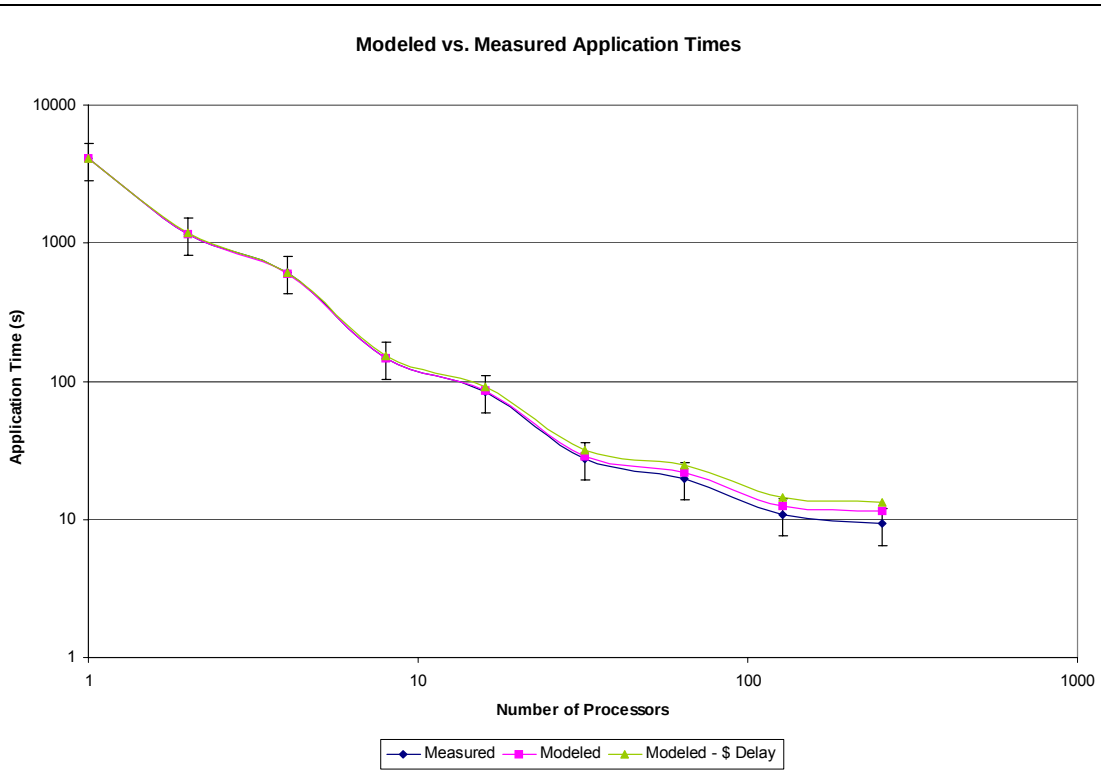

Figure 46 – ALC Class C Model Comparison

3 – Keck Cluster Analysis

Following are sample graphs from the Keck Cluster showing the measured values (blue), the model excluding the switch delay (magenta), and the model including the switch delay (green).

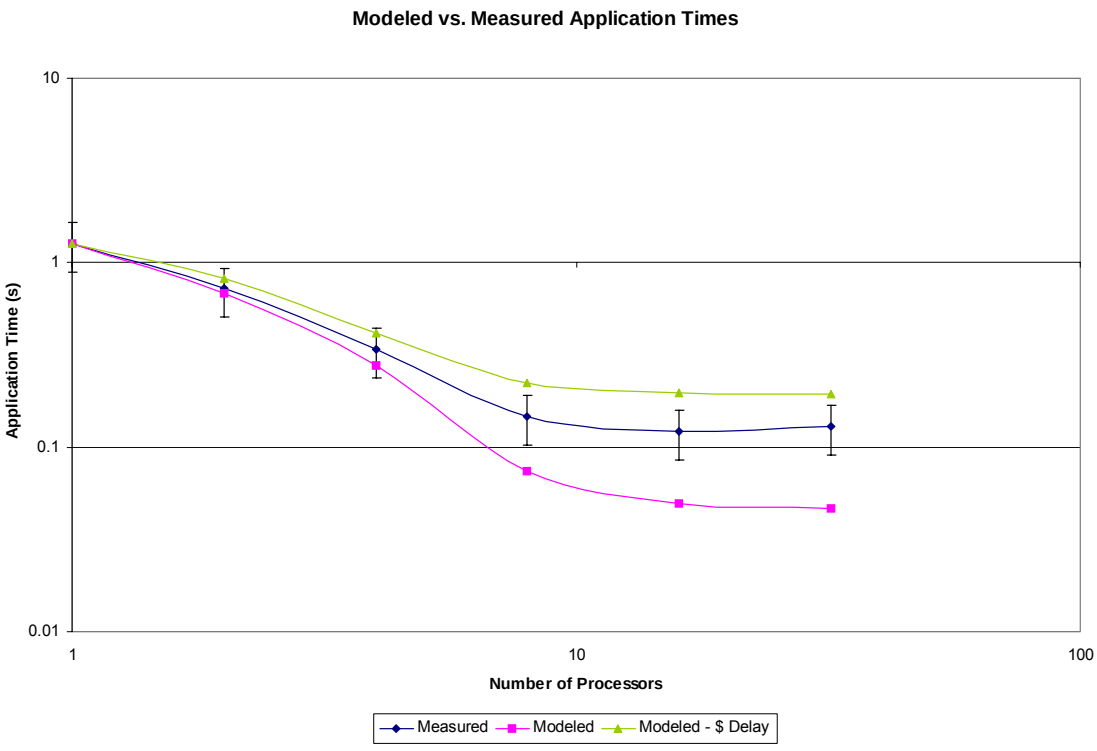

Figure 47 – Keck Cluster Class S Model Comparison

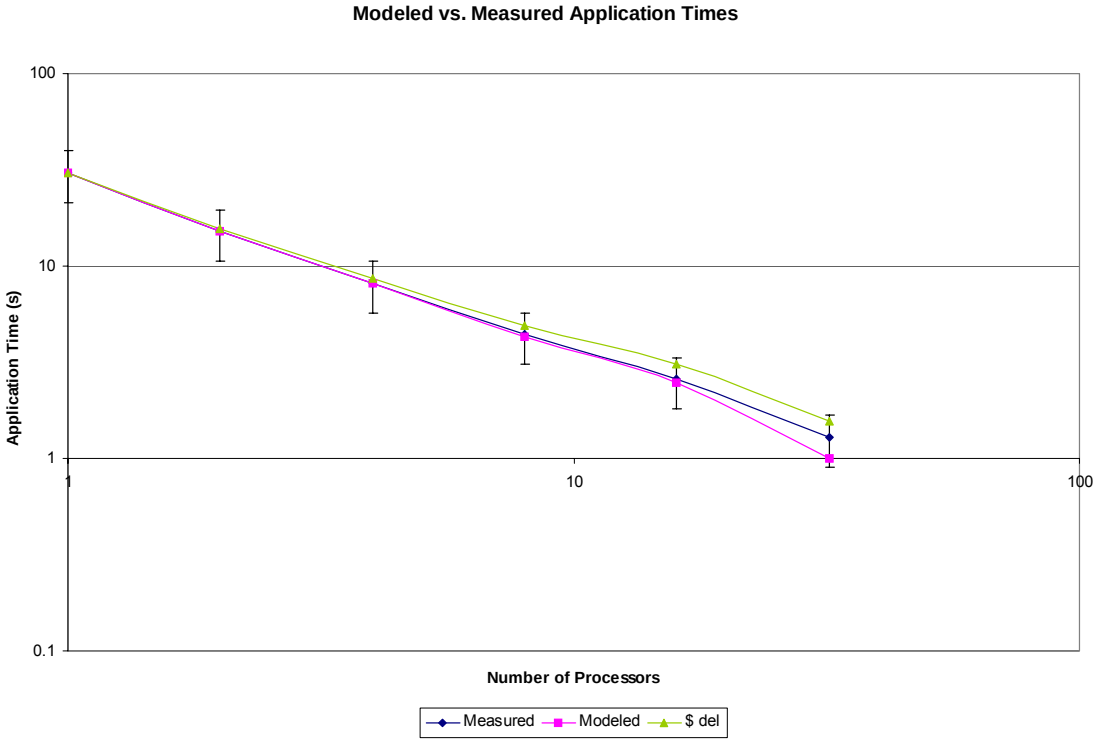

Figure 48 – Keck Cluster Class A Model Comparison

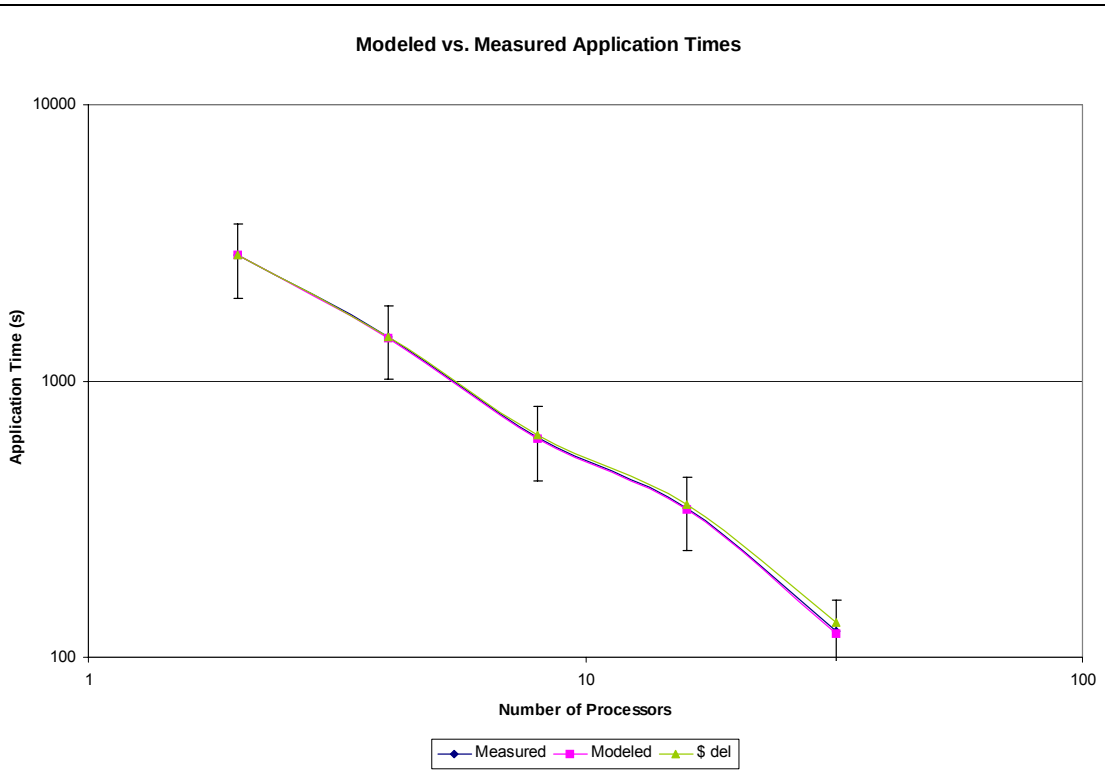

Figure 49 – Keck Cluster Class C Model Comparison

4 – Final Analysis

The above results indicate more research as to why MCR and ALC show better modeling without the switch delay and the Keck Cluster shows better modeling with the switch delay. One possibility is that the data used to determine bandwidth and latency consider the switch delay for MCR and ALC, therefore including the switching characteristics in the model separately is redundant, and the data for the Keck Cluster does not, thus necessitating the inclusion of an explicit switch delay in the model. This and other possibilities will also be explored in the near future.

E – Measure and Predict Additional Systems

While the research preceding is sufficient to provide a proof-of-concept for QNM modeling, the use of three test systems, each utilizing a Linux derivative, is insufficient to prove the universality of the concept, and its applicability to either other operating systems or different clusters. Therefore, the QNM modeling and data collection procedure needs to be performed on additional computers and under different POSIX derivatives. Candidates for new systems include BlueGene/L (Linux), Thunder (Linux), and Berg/NewBerg (AIX), all machines at LLNL. Ultimately, the modeling procedure should also be performed under non-POSIX systems as well.

F – Measure and Predict Additional Applications

As in Section E, above, the use of NPB-CG is sufficient for proof-of-concept testing, but is not so for proof of universal application. Additional testing is required to expand the QNM concept for more general use, beginning with application of QNM to the rest of the NBP suite, in particular FT and BT. Once the NBP suite is fully analyzed, testing

should continue with other programs, such as those mentioned in the benchmarking review in Chapter II.

G – Refine Model for Interconnect

The current model of the interconnect between computation nodes assumes a flat switching hierarchy and a single queueing node which incurs the total delay for the entire switching process as well as any overhead incurred by the MPI and interconnect protocol stacks. Continued examination of this switching model is necessary to improve accuracy in the QNM modeling process.

VI – Summary and Conclusion

Queueing Network Modeling is a possible solution to the modeling problems facing modern high-performance computing. When given proper values for machine, network, and program characteristics, QNM can accurately predict runtimes and performance on a given machine with a given problem within the typically expected accuracy of QNM, which is 10 – 30 %, as the data in Chapter IV clearly shows.

Apriori parameter prediction for the QNM model does show some level of difficulty, as does prediction of the movement of MPI from typical (decreasing runtimes as processors are added) to atypical (increasing run times as processors are added) behavior. However, the QNM method itself, when these hurdles are overcome, promises to be a very powerful tool in both the design and operation of high-performance computers. Possible uses of QNM are batch system scheduling, cost-benefit analysis, and better hardware/software engineering tools.

QNM is easy to understand, simple to parameterize (given above caveat about determining the parameters), and mathematically uncomplicated. The QNM algorithm itself is highly efficient and requires very little computation time to derive a solution. As research continues to make headway on the difficulties mentioned in Chapter V, QNM shows great promise to become one of the leading tools used for performance prediction and modeling of large-scale high-performance computers.

VII – Appendices

Appendix A – Latency and Bandwidth Data

1 – Keck Cluster

The Keck Cluster Latency and Bandwidth Curve in Figure 50 was constructed from data available for MPICH-GM with GM 1.x from the Myrinet website.[53] The Myrinet documentation reports the tests were conducted using the PALLAS MPI Benchmark Suite V2.2, MPI-1 part, Release 2.4.19. Figure 50 plots the results for the PingPong benchmark, which is the most similar to the LLNL LBW benchmark, while Table 13 gives raw data figures. We used the Myrinet values, as we had difficulty modifying the LLNL LBW configuration and makefile for the Keck Cluster.

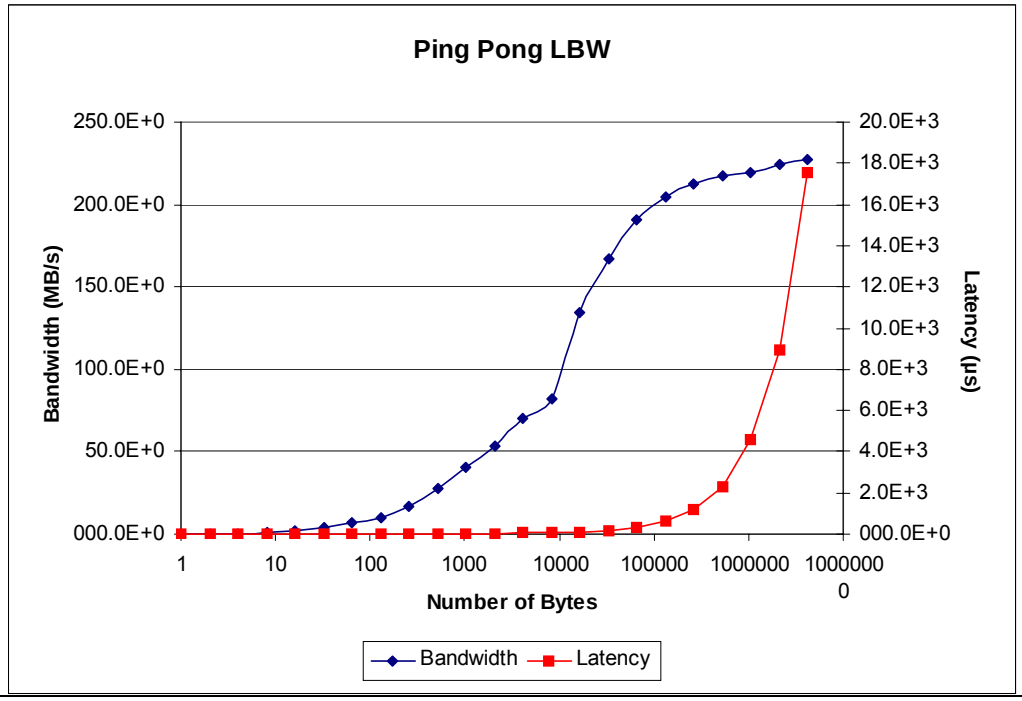

Figure 50 – Keck Cluster Latency and Bandwidth Curve

[53] www.myrinet.com/myrinet/performance

Number of Bytes	Latency (µs)	MBytes/s
0	8.05	0
1	8.14	0.12
2	8.14	0.23
4	8.18	0.47
8	8.29	0.92
16	8.61	1.77
32	8.77	3.48
64	9.03	6.76
128	12.08	10.1
256	14.83	16.45
512	17.89	27.29
1,024	24.31	40.17
2,048	36.88	52.95
4,096	55.67	70.17
8,192	95.27	82
16,384	116.47	134.15
32,768	186.97	167.14
65,536	327.91	190.6
131,072	609.82	204.98
262,144	1,174.19	212.91
524,288	2,303.11	217.1
1,048,576	4,561.19	219.24
2,097,152	8,898.06	224.77
4,194,304	17,573.37	227.62

Table 13 – Keck Cluster Latency and Bandwidth Raw Data

2 – MCR

Figure 51 and Figure 52 plot the results for the LBW benchmark on MCR, while Table 14, Table 15, and Table 16 give raw data figures for bandwidth and latency. All model calculations assume internode asynchronous communication, which is the case for the NAS-PB CG benchmark using one processor per node.

The primary peak in the intranode bandwidth indicates the transfer rate from data in the cache (cache hits); whereas the trailing plateau represents transfer rates from main memory (cache misses). Internode communication is not nearly as influenced by caching effects, and thus does not show the peaks noticed in intranode communication. Bandwidth for asynchronous communication between nodes is much higher for large

messages than bandwidth for synchronous communications, as much of the MPI time that is taken making the non-blocking receive call can be processed in parallel with the send, resulting in faster effective transfer rates. With synchronous communication, this call overhead must be processed serially, and results in less data transfer.

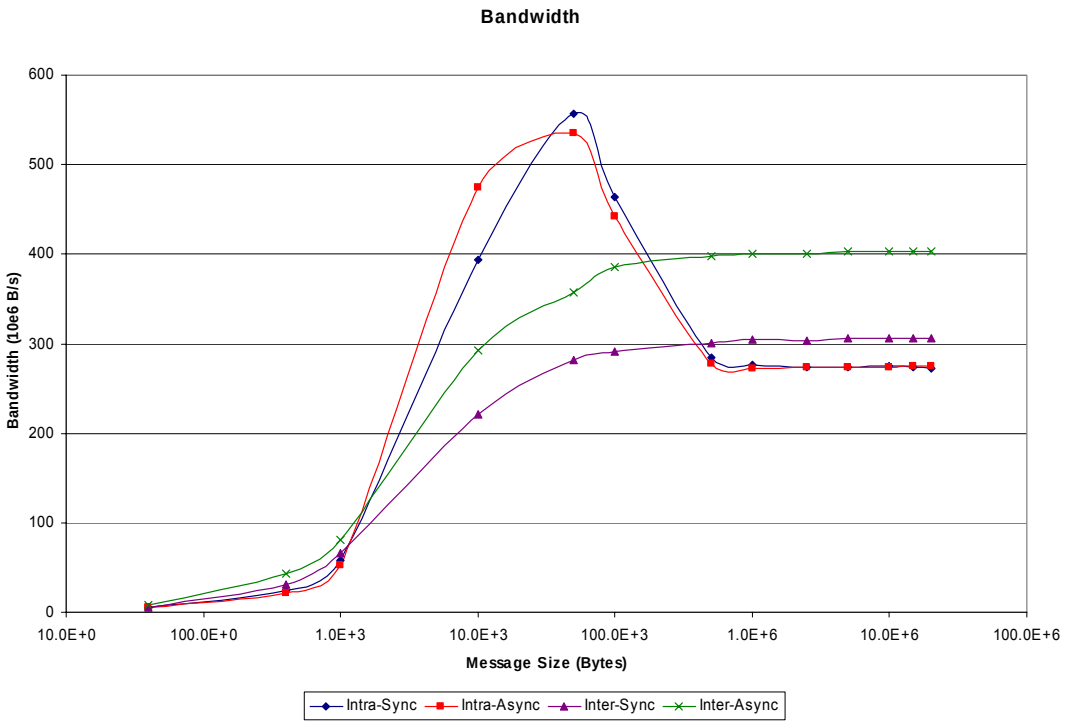

Figure 51 – MCR Bandwidth Curve

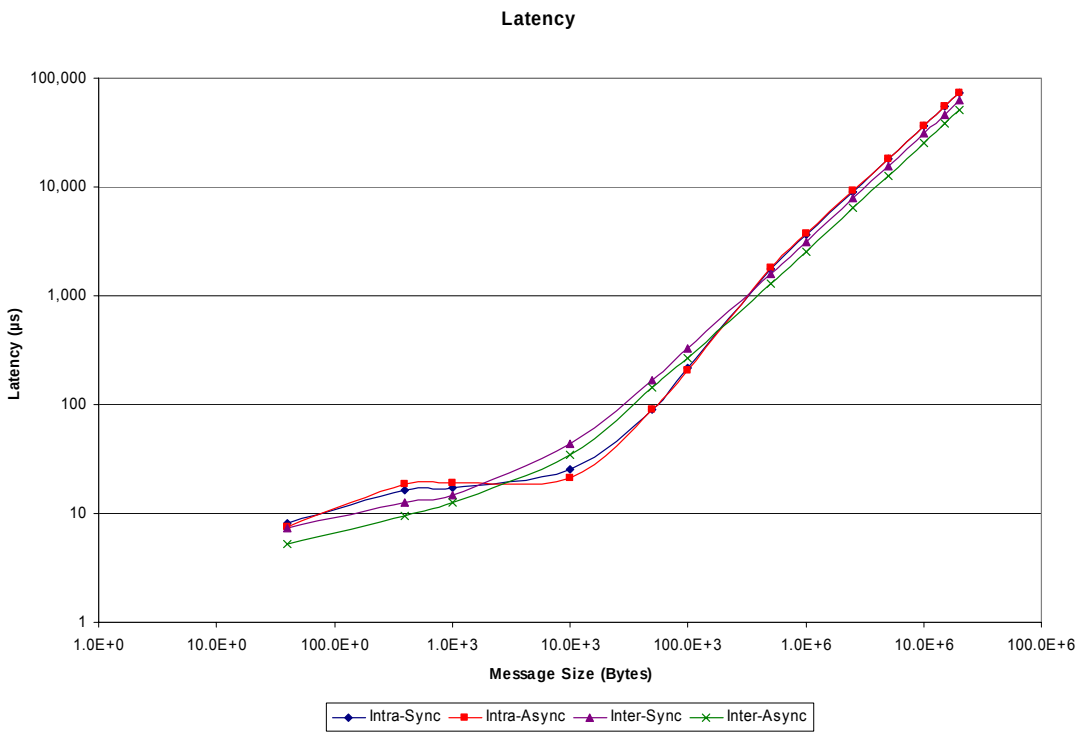

Figure 52 – MCR Latency Curve

Message Size	Bandwidth (10e6 B/s)				Latency (µs)			
	Intra		Inter		Intra		Inter	
	Sync	Async	Sync	Async	Sync	Async	Sync	Async
40	3.9726	4.811	4.52	5.367	10.156	8.284	9.248	7.462
400	21.2054	20.8654	26.8278	31.5842	19.018	19.19	15.366	12.756
1,000	51.3228	51.1248	58.4316	66.9286	19.484	19.464	17.472	15.1
10,000	367.3934	420.6224	217.3698	262.9602	27.326	23.742	46.674	38.276
50,000	541.001	516.7798	292.6886	349.9428	95.33	94.266	172.334	145.336
100,000	429.87	426.9608	305.3328	377.29	245.994	226.008	330.388	270.578
500,000	290.3044	278.7632	317.5568	389.7146	1,768.266	1,818.076	1,586.674	1,292.112
1,000,000	281.9118	275.782	321.4458	394.5908	3,596.704	3,664.92	3,135.946	2,551.376
2,500,000	281.3806	277.3858	320.2202	392.3956	8,999.088	9,163.904	7,866.54	6,386.956
5,000,000	279.2804	276.9606	322.594	394.3502	18,062.88	18,170.846	15,595.29	12,702.79
10,000,000	280.022	277.765	322.8158	394.317	36,238.014	36,583.034	31,147.338	25,338.628
15,000,000	281.3414	278.8056	322.9122	395.3364	54,063.476	55,604.676	46,708.204	37,962.298
20,000,000	281.4702	278.1846	322.9528	395.4602	71,930.252	72,471.956	62,278.82	50,676.188

Table 14 – MCR Bandwidth and Latency Data with mpiP

Message Size	Bandwidth (10e6 B/s)				Latency (µs)			
	Intra		Inter		Intra		Inter	
	Sync	Async	Sync	Async	Sync	Async	Sync	Async
40	0.08	0.0499	0.0403	0.0549	0.0586	0.0602	0.1763	0.1594
400	0.1465	0.1365	0.3491	0.5231	0.2119	0.088	0.2959	0.6359
1,000	0.4411	0.2852	0.6321	0.2549	0.0945	0.0716	0.1392	0.6904
10,000	3.6091	6.7215	0.8565	4.9315	0.194	0.2643	0.8376	1.8696
50,000	20.5529	30.4183	0.7395	4.6366	3.5088	3.9649	2.8005	6.6513
100,000	15.1763	15.1993	0.9421	2.2082	9.5574	13.8706	5.2579	13.6717
500,000	4.2409	5.3838	0.9253	4.8311	17.1772	37.7537	26.1131	66.6365
1,000,000	2.9619	2.4008	0.8293	5.2299	14.7897	34.077	51.919	126.925
2,500,000	4.1393	1.6986	0.7837	5.2014	47.0401	88.984	127.385	285.3246
5,000,000	3.2725	2.5598	0.8914	4.8161	86.3938	85.5093	217.7568	526.7418
10,000,000	3.3342	2.5839	0.8104	5.5202	251.3583	473.3553	379.5457	1,051.9095
15,000,000	3.6915	1.8167	0.7744	5.1362	446.9563	1,509.5416	559.5785	1,545.5308
20,000,000	3.592	3.0111	0.7757	5.2378	384.4657	736.5662	769.6137	1,949.7053

Table 15 – MCR Bandwidth and Latency Standard Deviation with mpiP

Message Size	Bandwidth (10e6 B/s)				Latency (µs)			
	Intra		Inter		Intra		Inter	
	Sync	Async	Sync	Async	Sync	Async	Sync	Async
40	4.9790	5.2116	5.4722	7.8422	8.0340	7.4300	7.2620	5.1800
400	23.9704	21.2372	30.8588	42.7874	16.4560	18.7240	12.7820	9.6180
1,000	57.9796	53.1218	66.2044	80.3926	17.1000	18.9680	14.8240	12.7500
10,000	394.3124	475.0980	220.6384	292.8932	25.0880	21.0780	43.4780	34.6900
50,000	556.4770	534.8658	281.1688	357.0646	89.7320	89.4880	169.2960	141.9780
100,000	463.6230	442.6388	291.8298	385.8690	216.0260	206.3560	325.2840	267.3200
500,000	284.7436	277.3154	301.1294	397.4968	1,744.9640	1,831.3920	1,574.8600	1,287.5560
1,000,000	276.3436	272.8988	304.4656	400.9956	3,600.9660	3,696.4320	3,115.0380	2,541.5640
2,500,000	273.7220	274.1934	303.2010	400.0838	9,026.6940	9,227.0960	7,821.3160	6,370.4100
5,000,000	273.7434	274.1276	305.4150	402.7298	18,169.2860	18,268.9700	15,525.4200	12,678.5020
10,000,000	274.4934	274.2700	305.6090	403.2826	36,321.9920	36,477.1960	31,031.1220	25,351.2540
15,000,000	273.2296	274.4400	305.4950	403.7056	54,480.4920	54,859.9200	46,523.3720	38,008.9560
20,000,000	272.6774	274.7892	305.4882	402.6148	73,002.8640	73,425.4520	62,034.7640	50,747.5640

Table 16 – MCR Bandwidth and Latency Data without mpiP

3 – ALC

Figure 53 and Figure 54 plot the results for the LBW benchmark on ALC, while Table 17, Table 18, and Table 19 gives raw data figures for bandwidth and latency. All model calculations assume internode asynchronous communication, which is the case for the NAS-PB CG benchmark using one processor per node.

Notice that the bandwidth curves are similar to those produced by MCR. MCR and ALC are very similar machines, and typically produce very similar results under the same test conditions. This is due to the same caching and parallelization of non-blocking calls, as explained above for MCR, and is what one would expect for a closely related architecture.

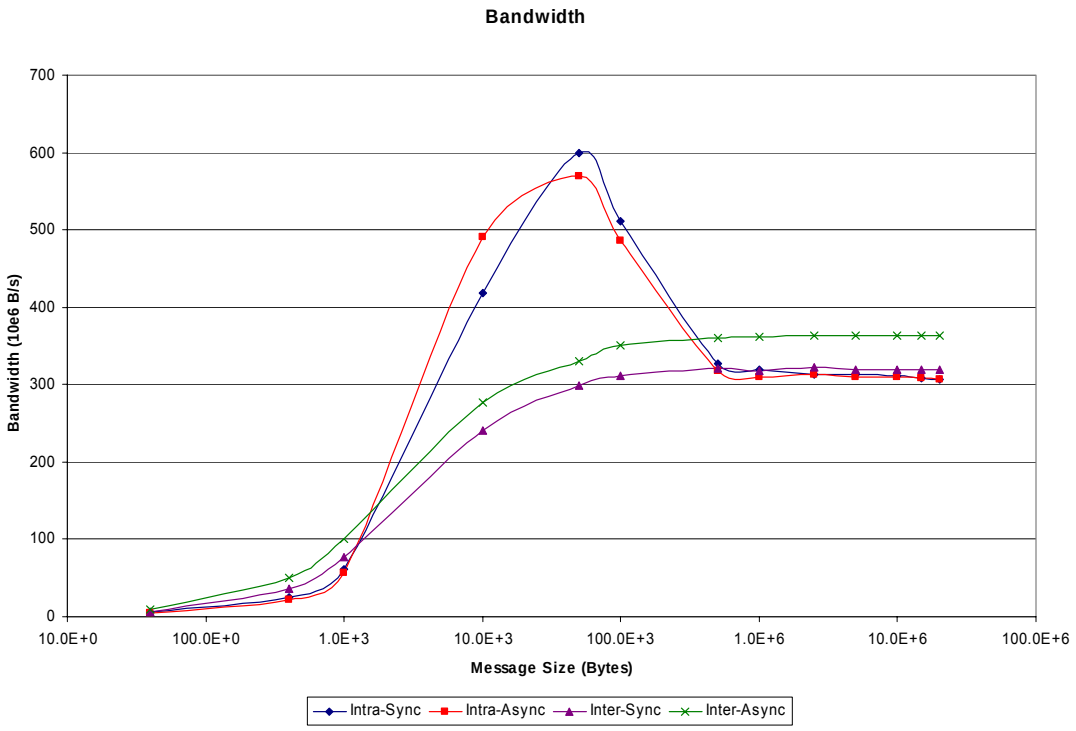

Figure 53 – ALC Bandwidth Curve

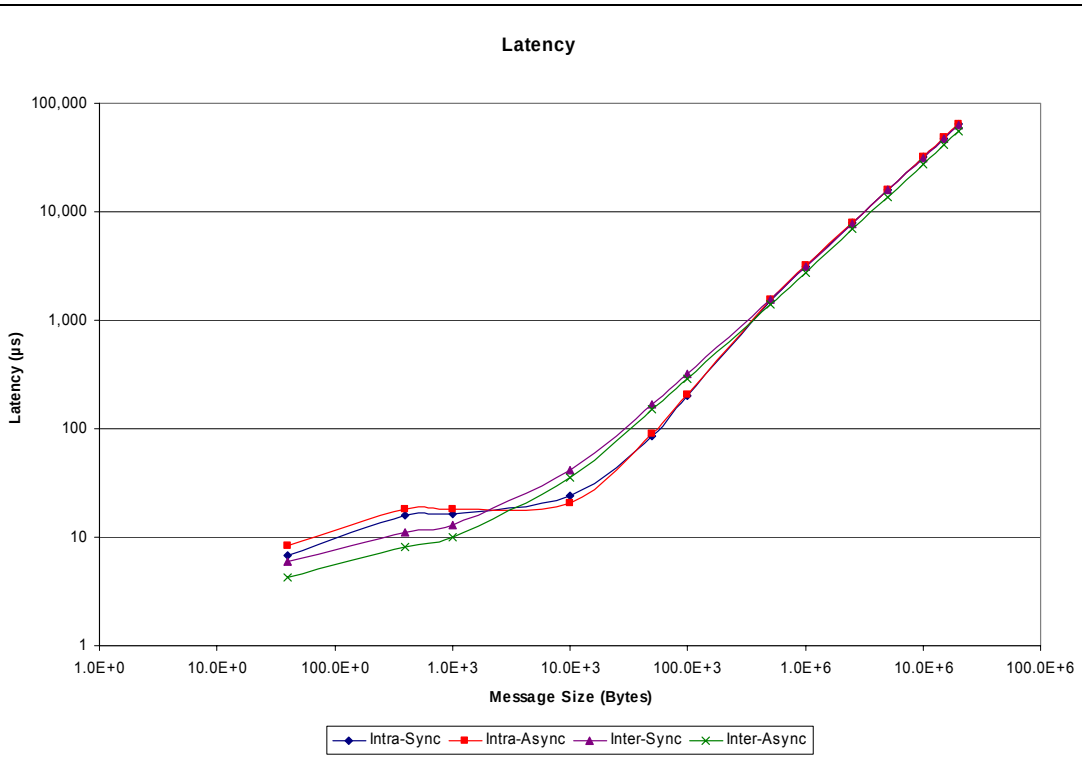

Figure 54 – ALC Latency Curve

Message Size	Bandwidth (10e6 B/s)				Latency (µs)			
	Intra		Inter		Intra		Inter	
	Sync	Async	Sync	Async	Sync	Async	Sync	Async
40	5.9764	4.9004	6.6976	9.4790	6.7900	8.3400	5.9880	4.2120
400	25.1672	22.2102	36.3244	49.5768	15.7440	18.0320	11.0480	8.0980
1,000	61.4002	55.9302	77.3420	101.1014	16.3920	18.1380	13.0820	9.8840
10,000	419.0350	490.8522	241.1840	277.2746	23.8720	20.4620	41.6340	35.8120
50,000	599.6436	570.2200	298.9056	330.1730	85.0620	90.1880	167.5600	151.2780
100,000	511.2758	486.5640	311.5230	350.6030	202.9900	204.6100	321.7060	286.6160
500,000	327.8804	317.1758	320.7300	360.6768	1,523.0360	1,564.1020	1,562.5880	1,389.5660
1,000,000	318.6446	310.0188	318.1268	362.5292	3,141.6340	3,204.3580	3,155.8840	2,768.1380
2,500,000	312.9746	313.7038	322.9082	363.2792	7,896.4460	7,981.0700	7,760.1120	6,884.3620
5,000,000	312.2842	309.4832	319.3196	363.0602	15,909.9160	16,016.8440	15,745.2580	13,759.4960
10,000,000	310.8564	309.6742	319.4644	363.3290	32,148.1000	31,944.8020	31,471.9480	27,492.1060
15,000,000	308.8984	307.9560	319.6816	363.5872	47,715.7280	48,122.0200	47,196.8900	41,216.3540
20,000,000	307.2910	307.1802	319.8050	363.9454	64,053.6120	64,898.2160	62,880.0960	54,934.4660

Table 17 – ALC Bandwidth and Latency Data with mpiP

Message Size	Bandwidth (10e6 B/s)				Latency (µs)			
	Intra		Inter		Intra		Inter	
	Sync	Async	Sync	Async	Sync	Async	Sync	Async
40	0.032	0.0503	0.04	0.1179	0.0823	0.1475	0.0495	0.039
400	0.1699	0.0799	0.393	1.2765	0.2509	0.1205	0.0919	0.0799
1,000	0.3682	0.3505	0.8688	4.3586	0.1891	0.2298	0.1128	0.1193
10,000	1.333	14.7197	2.9456	0.2176	0.1568	0.1963	0.0559	0.2741
50,000	17.8992	11.8619	3.6162	3.6216	2.4068	7.5937	0.2303	1.0505
100,000	38.3299	28.7826	5.3652	3.2134	14.1377	20.0818	0.7493	2.808
500,000	4.9798	4.3932	4.3423	1.1353	15.7547	27.7759	5.79	11.7985
1,000,000	1.7375	1.9569	4.5587	1.6141	40.5172	22.3529	14.1584	18.8678
2,500,000	2.8588	2.4972	4.2779	1.4647	223.3009	166.0422	30.9485	57.9774
5,000,000	1.5366	2.7166	4.4801	1.632	103.8533	197.1605	89.7334	122.5853
10,000,000	2.8239	2.0616	2.5979	1.3202	835.4868	364.3385	187.661	277.9518
15,000,000	3.0522	5.1042	2.5242	1.2745	1,371.6671	302.5326	282.0873	410.9768
20,000,000	3.2741	6.7031	2.9886	1.3407	2,104.2451	1,791.2003	358.355	473.5005

Table 18 – ALC Bandwidth and Latency Standard Deviation with mpiP

Message Size	Bandwidth (10e6 B/s)				Latency (µs)			
	Intra		Inter		Intra		Inter	
	Sync	Async	Sync	Async	Sync	Async	Sync	Async
40	4.7162	5.5048	5.1944	6.3286	8.428	7.34	7.69	6.322
400	22.6806	21.9986	30.369	36.5076	17.604	18.232	13.05	10.784
1,000	55.2124	53.9318	67.2184	80.013	18.142	18.596	14.736	12.364
10,000	388.5484	448.0326	225.7582	260.151	25.544	22.494	43.564	38.47
50,000	584.6418	589.1928	295.113	325.0722	86.936	93.344	167.796	152.11
100,000	468.8678	469.3568	304.9666	347.3834	199.63	216.192	322.374	288.846
500,000	310.8074	305.0264	316.6902	361.2918	1,609.666	1,606.944	1,558.15	1,385.78
1,000,000	307.2104	305.099	318.4326	362.5274	3,260.692	3,259.574	3,121.812	2,768.038
2,500,000	306.183	302.6926	319.3902	363.805	8,233.526	8,113.64	7,738.196	6,883.364
5,000,000	306.943	305.4298	319.0074	364.227	16,214.356	16,023.622	15,548.572	13,793.11
10,000,000	308.9394	305.952	320.674	364.5834	32,857.978	32,053.816	31,071.87	27,559.196
15,000,000	307.8658	303.582	320.7078	364.8706	49,314.166	48,447.148	46,596.196	41,270.232
20,000,000	305.7862	305.112	320.5504	365.2064	66,837.706	64,534.358	62,114.756	54,993.356

Table 19 – ALC Bandwidth and Latency Data without mpiP

Appendix B – Sample NPB Spreadsheet

In this section, we present a sample of the NPB spreadsheet, which contains actual data collected from MCR and is used to create the graphs and results presented in Chapter IV. Spreadsheets for ALC and the Keck Cluster are similar, and work in the same manner. The headings highlighted in black denote the major sections of the spreadsheet. The Collected Values section contains mpiP and NAS-PB data collected from the machines during the actual execution of the application, as provided by the mpiPfilter program, and is the starting point for analysis. The Analysis Views section contains various analyses of the collected data from different viewpoints of the system. The Utilization Views section contains information about the utilization of various system resources. The Network View section contains analysis of the network resource utilization. The Model Views section contains two related subsections, Model Inputs, which collects the results of the analyses in the previous sections as inputs for the QNM model, and Model Outputs, which contain the results from running the QNM model on the provided inputs. The Validation View section contains error analysis information used to validate the model. The Graphical View section contains data from the previous sections, arranged to make graphical display easier.

Label	Symbol	Derivation	Unit	Type	1	2	4	8	16	32	64	128	256	512
Collected Values														
Parameters					**No Switch Delay**									
Number of CPUs	P	-	CPU	IO	1	2	4	8	16	32	64	128	256	512
Application	-	-	Text	I	CG S	CG S	CG S	CG S	CG S	CG S	CG S	CG S	CG S	CG S
Machine	-	-	Text	I	mcr2	mcr2	mcr2	mcr2	mcr2	mcr2	mcr2	mcr2	mcr2	mcr2
Run Date	-	-	Date	I	2/25/06	2/25/06	2/25/06	2/26/06	2/26/06	2/25/06	2/25/06	2/25/06	2/25/06	2/25/06
mpiP Collector PID	-	-	#	I	11479	11815	21275	16955	16400	28358	4400	11260	26277	26042
mpiP														
Aggregate Application Time	App_Time	-	s	I	216.8E-3	304.0E-3	375.8E-3	725.8E-3	1.388E+0	3.554E+0	7.516E+0	20.06E+0	44.28E+0	180.2E+0
Aggregate MPI Time	MPI_Time	-	s	I	55.0E-6	94.62E-3	218.8E-3	497.4E-3	1.026E+0	2.762E+0	6.04E+0	16.68E+0	37.78E+0	164.6E+0
Aggregate MPI_WAIT	MPI_Wait	-	s	A	000.0E+0	19.126E-3	71.193E-3	123.213E-3	240.903E-3	907.787E-3	2.192E+0	7.93E+0	20.49E+0	124.485E+0
Number of Messages Sent	M	-	msg	A	1.0E+0	3.152E+3	6.304E+3	22.088E+3	44.176E+3	126.272E+3	252.544E+3	656.768E+3	1.314E+6	3.234E+6
Average Sent Message Size	L	-	B	A	8.0E+0	2.776E+3	2.776E+3	1.191E+3	1.191E+3	558.4E+0	558.4E+0	271.083E+0	271.082E+0	134.802E+0
Linux Time														
Average CPU Utilization	U^{*}_{CPU}	-	#	A	84.00%	87.70%	81.30%	81.65%	80.36%	79.41%	79.14%	72.57%	66.19%	60.68%
Average Elapsed Time	-	-	s	A	428.0E-3	323.0E-3	343.5E-3	318.75E-3	344.125E-3	740.562E-3	1.047E+0	1.321E+0	1.592E+0	2.721E+0
NAS-PB														
Elapsed Time	-	-	s	I	214.0E-3	152.0E-3	94.0E-3	90.0E-3	88.0E-3	112.0E-3	116.0E-3	156.0E-3	174.0E-3	350.0E-3
Mop/s	-	-	Mop/s	I	310.826E+0	440.75E+0	715.166E+0	736.326E+0	769.882E+0	601.456E+0	569.858E+0	427.014E+0	387.826E+0	210.166E+0
Mop/s/process	-	-	Mop/s	I	310.826E+0	220.374E+0	178.792E+0	92.042E+0	48.116E+0	18.794E+0	8.904E+0	3.334E+0	1.516E+0	410.0E-3
Network Info.														
Bandwidth	BW	(Linear Interpolation)	B/s	I	1.568E+6	122.319E+6	122.319E+6	84.908E+6	84.908E+6	52.715E+6	52.715E+6	30.273E+6	30.273E+6	17.045E+6
Latency	Lat	-	s	I	5.18E-6	5.18E-6	5.18E-6	5.18E-6	5.18E-6	5.18E-6	5.18E-6	5.18E-6	5.18E-6	5.18E-6

Analysis Views														
Aggregate														
Application Time	App_Time	-	s	I	216.8E-3	304.0E-3	375.8E-3	725.8E-3	1.388E+0	3.554E+0	7.516E+0	20.06E+0	44.28E+0	180.2E+0
MPI Time	MPI_Time	-	s	I	55.0E-6	94.62E-3	218.8E-3	497.4E-3	1.026E+0	2.762E+0	6.04E+0	16.68E+0	37.78E+0	164.6E+0
Non-MPI Time	Non_MPI	App_Time - MPI_Time	s	C	216.745E-3	209.38E-3	157.0E-3	228.4E-3	362.4E-3	792.0E-3	1.476E+0	3.38E+0	6.5E+0	15.6E+0
MPI_WAIT	MPI_Wait	-	s	A	000.0E+0	19.126E-3	71.193E-3	123.213E-3	240.903E-3	907.787E-3	2.192E+0	7.93E+0	20.49E+0	124.485E+0
MPI Active Time	MPI_Active	MPI_Time - MPI_Wait	s	C	55.0E-6	75.494E-3	147.607E-3	374.187E-3	784.697E-3	1.854E+0	3.848E+0	8.75E+0	17.29E+0	40.115E+0
Per CPU														
Application Time	AT*	App_Time / P	s	CIV	216.8E-3	152.0E-3	93.95E-3	90.725E-3	86.75E-3	111.063E-3	117.438E-3	156.719E-3	172.969E-3	351.953E-3
MPI Time	MT*	MPI_Time / P	s	CIV	55.0E-6	47.31E-3	54.7E-3	62.175E-3	64.1E-3	86.313E-3	94.375E-3	130.313E-3	147.578E-3	321.484E-3
Non-MPI Time	-	(App_Time - MPI_Time) / P	s	C	216.745E-3	104.69E-3	39.25E-3	28.55E-3	22.65E-3	24.75E-3	23.063E-3	26.406E-3	25.391E-3	30.469E-3
MPI_WAIT	WT*	MPI_Wait / P	s	C	000.0E+0	9.563E-3	17.798E-3	15.402E-3	15.056E-3	28.368E-3	34.256E-3	61.952E-3	80.041E-3	243.134E-3
MPI Active Time	-	(MPI_Time - MPI_Wait) / P	s	C	55.0E-6	37.747E-3	36.902E-3	46.773E-3	49.044E-3	57.944E-3	60.119E-3	68.361E-3	67.537E-3	78.35E-3
Per Sent Message														
Application Time	-	App_Time / M	s	C	216.8E-3	96.447E-6	59.613E-6	32.859E-6	31.42E-6	28.146E-6	29.761E-6	30.544E-6	33.711E-6	55.724E-6
MPI Time	-	MPI_Time / M	s	C	55.0E-6	30.019E-6	34.708E-6	22.519E-6	23.216E-6	21.873E-6	23.917E-6	25.397E-6	28.762E-6	50.9E-6
Non-MPI Time	MsgCompute	(App_Time - MPI_Time) / M	s	CO	216.745E-3	66.428E-6	24.905E-6	10.34E-6	8.204E-6	6.272E-6	5.845E-6	5.146E-6	4.948E-6	4.824E-6
MPI_WAIT	-	MPI_Wait / M	s	C	000.0E+0	6.068E-6	11.293E-6	5.578E-6	5.453E-6	7.189E-6	8.681E-6	12.074E-6	15.599E-6	38.495E-6
MPI Active Time	-	(MPI_Time - MPI_Wait) / M	s	C	55.0E-6	23.951E-6	23.415E-6	16.941E-6	17.763E-6	14.684E-6	15.235E-6	13.323E-6	13.163E-6	12.405E-6
Per CPU per Sent Message														
Application Time	-	App_Time / (P * M)	s	C	216.8E-3	48.223E-6	14.903E-6	4.107E-6	1.964E-6	879.55E-9	465.018E-9	238.621E-9	131.682E-9	108.836E-9
MPI Time	-	MPI_Time / (P * M)	s	C	55.0E-6	15.01E-6	8.677E-6	2.815E-6	1.451E-6	683.544E-9	373.697E-9	198.415E-9	112.352E-9	99.414E-9
Non-MPI Time	-	(App_Time - MPI_Time) / (P * M)	s	C	216.745E-3	33.214E-6	6.226E-6	1.293E-6	512.722E-9	196.005E-9	91.321E-9	40.206E-9	19.33E-9	9.422E-9
MPI_WAIT	-	MPI_Wait / (P * M)	s	CV	000.0E+0	3.034E-6	2.823E-6	697.283E-9	340.829E-9	224.66E-9	135.644E-9	94.328E-9	60.935E-9	75.185E-9
MPI Active Time	CPUMsgActive	(MPI_Time - MPI_Wait) / (P * M)	s	CO	55.0E-6	11.976E-6	5.854E-6	2.118E-6	1.11E-6	458.884E-9	238.054E-9	104.087E-9	51.417E-9	24.229E-9

Utilization Views														
Per CPU														
Busy Time	-	(App_Time * U_{CPU}) / P	s	C	182.112E-3	133.304E-3	76.381E-3	74.077E-3	69.714E-3	88.198E-3	92.937E-3	113.729E-3	114.482E-3	213.57E-3
Idle Time	-	(App_Time * (1 - U_{CPU})) / P	s	C	34.688E-3	18.696E-3	17.569E-3	16.648E-3	17.036E-3	22.865E-3	24.5E-3	42.99E-3	58.486E-3	138.384E-3
Per Sent Message														
Busy Time	-	(App_Time * U_{CPU}) / M	s	C	182.112E-3	84.584E-6	48.465E-6	26.83E-6	25.25E-6	22.351E-6	23.552E-6	22.165E-6	22.312E-6	33.814E-6
Idle Time	-	(App_Time * (1 - U_{CPU})) / M	s	C	34.688E-3	11.863E-6	11.148E-6	6.03E-6	6.17E-6	5.794E-6	6.209E-6	8.378E-6	11.399E-6	21.91E-6
Per CPU per Sent Message														
Busy Time	-	(App_Time * U_{CPU}) / (P * M)	s	C	182.112E-3	42.292E-6	12.116E-6	3.354E-6	1.578E-6	698.472E-9	368.004E-9	173.164E-9	87.156E-9	66.043E-9
Idle Time	-	(App_Time * (1 - U_{CPU})) / (P * M)	s	C	34.688E-3	5.931E-6	2.787E-6	753.714E-9	385.629E-9	181.077E-9	97.014E-9	65.457E-9	44.526E-9	42.793E-9
Network View														
Per Network Switch														
Switch Delay	D_0	L/BW + Lat	s	CO	10.281E-6	27.872E-6	27.872E-6	19.21E-6	19.21E-6	15.773E-6	15.773E-6	14.134E-6	14.134E-6	13.089E-6
Model View														
Model Inputs														
Customers	N	P	#	B	1.0E+0	2.0E+0	4.0E+0	8.0E+0	16.0E+0	32.0E+0	64.0E+0	128.0E+0	256.0E+0	512.0E+0
Centers	K	P + 2	#	B	3.0E+0	4.0E+0	6.0E+0	10.0E+0	18.0E+0	34.0E+0	66.0E+0	130.0E+0	258.0E+0	514.0E+0
Switch Delay	D_0	L/BW + Lat	s	B	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0
CPU Service Demand	D_k	(MPI_Time - MPI_Wait) / (P * M) = CPU Message Active	s	B	55.0E-6	11.976E-6	5.854E-6	2.118E-6	1.11E-6	458.884E-9	238.054E-9	104.087E-9	51.417E-9	24.229E-9
Computation Delay	D_{P+1}	(App_Time - MPI_Time) / M = MsgCompute	s	B	216.745E-3	66.428E-6	24.905E-6	10.34E-6	8.204E-6	6.272E-6	5.845E-6	5.146E-6	4.948E-6	4.824E-6
Model Outputs														
System Response Time	R	-	s	R	216.8E-3	93.552E-6	57.83E-6	37.868E-6	39.031E-6	32.393E-6	33.439E-6	29.364E-6	28.98E-6	27.431E-6
Switch Response Time	R_0	-	s	R	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0
CPU Response Time	R_k	-	s	R	55.0E-6	13.562E-6	8.231E-6	3.441E-6	1.927E-6	816.266E-9	431.168E-9	189.203E-9	93.873E-9	44.154E-9
Computation Response Time	R_{P+1}	-	s	R	216.745E-3	66.428E-6	24.905E-6	10.34E-6	8.204E-6	6.272E-6	5.845E-6	5.146E-6	4.948E-6	4.824E-6
System Throughput	X	-	msg/s	R	4.613E+0	21.378E+3	69.168E+3	211.259E+3	409.927E+3	987.878E+3	1.914E+6	4.359E+6	8.834E+6	18.665E+6
Switch Utilization	U_0	-	#	R	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0
CPU MPI Utilization	U_k	-	#	R	253.69E-6	256.019E-3	404.887E-3	447.36E-3	455.095E-3	453.321E-3	455.615E-3	453.716E-3	454.198E-3	452.229E-3
Total Computation Utilization	U_{P+1}	-	#	R	999.746E-3	1.42E+0	1.723E+0	2.185E+0	3.363E+0	6.196E+0	11.186E+0	22.433E+0	43.713E+0	90.041E+0

Validation View													
Application (Wall Clock) Time													
App Time - Observed (Wall Clock)	AT^*	App_Time / P	s	CI	216.8E-3	152.0E-3	93.95E-3	90.725E-3	86.75E-3	111.063E-3	117.438E-3	156.719E-3	172.969E-3	351.953E-3
App Time - Model	AT	(R * M) / P	s	CR	216.8E-3	147.439E-3	91.14E-3	104.554E-3	107.765E-3	127.821E-3	131.951E-3	150.669E-3	148.696E-3	173.254E-3
Relative Error	E_{AT}	$(AT - AT^*) / AT^*$	%	C	0.0%	-3.0%	-3.0%	15.2%	24.2%	15.1%	12.4%	-3.9%	-14.0%	-50.8%
MPI Active Time													
MPI Time - Observed	MT^*	MPI_Time / P	s	CI	55.0E-6	47.31E-3	54.7E-3	62.175E-3	64.1E-3	86.313E-3	94.375E-3	130.313E-3	147.578E-3	321.484E-3
MPI Time - Model	MT	$(R_k * M) + (R_0 * M) / P$	s	CR	55.0E-6	42.749E-3	51.89E-3	76.004E-3	85.115E-3	103.071E-3	108.889E-3	124.262E-3	123.306E-3	142.785E-3
Relative Error	E_{MT}	$(MT - MT^*) / MT^*$	%	C	0.0%	-9.6%	-5.1%	22.2%	32.8%	19.4%	15.4%	-4.6%	-16.4%	-55.6%
MPI Wait Time													
MPI_Wait Time - Estimated	WT^*	MPI_Wait / P	s	C	000.0E+0	9.563E-3	17.798E-3	15.402E-3	15.056E-3	28.368E-3	34.256E-3	61.952E-3	80.041E-3	243.134E-3
MPI_Wait Time - Model	WT	$(R_k - D_k) * M + (R_0 * M) / P$	s	CR	429.573E-15	5.002E-3	14.989E-3	29.231E-3	36.072E-3	45.127E-3	48.77E-3	55.902E-3	55.768E-3	64.435E-3
Relative Error	E_{WT}	$(WT - WT^*) / WT^*$	%	C	#DIV/0!	-47.7%	-15.8%	89.8%	139.6%	59.1%	42.4%	-9.8%	-30.3%	-73.5%
Throughput													
Throughput - Observed	X^*	M / AT^*	msg/s	C	4.613E+0	20.737E+3	67.1E+3	243.461E+3	509.233E+3	1.137E+6	2.15E+6	4.191E+6	7.594E+6	9.188E+6
Throughput - Model	X	-	msg/s	R	4.613E+0	21.378E+3	69.168E+3	211.259E+3	409.927E+3	987.878E+3	1.914E+6	4.359E+6	8.834E+6	18.665E+6
Relative Error	E_X	$(X - X^*) / X^*$	%	C	0.0%	3.1%	3.1%	-13.2%	-19.5%	-13.1%	-11.0%	4.0%	16.3%	103.1%
CPU													
CPU Utilization - Observed	U^*_{CPU}	-	#	A	840.0E-3	877.0E-3	813.0E-3	816.5E-3	803.625E-3	794.125E-3	791.375E-3	725.688E-3	661.867E-3	606.813E-3
CPU Utilization - Model	U_{CPU}	$(U_{P+1} / P) + U_k$	#	C	1.0E+0	966.076E-3	835.541E-3	720.424E-3	665.274E-3	646.95E-3	630.395E-3	628.977E-3	624.953E-3	628.091E-3
Relative Error	E_{CPU}	$(U - U^*) / U^*$	%	C	19.0%	10.2%	2.8%	-11.8%	-17.2%	-18.5%	-20.3%	-13.3%	-5.6%	3.5%
Graphical View													
Measured Component Time													
MPI Wait	-	MPI_Wait	s	G	000.0E+0	19.126E-3	71.193E-3	123.213E-3	240.903E-3	907.787E-3	2.192E+0	7.93E+0	20.49E+0	124.485E+0
MPI Active	-	MPI_Time - MPI_Wait	s	G	55.0E-6	75.494E-3	147.607E-3	374.187E-3	784.697E-3	1.854E+0	3.848E+0	8.75E+0	17.29E+0	40.115E+0
Computation	-	Non-MPI Time	s	G	216.745E-3	209.38E-3	157.0E-3	228.4E-3	362.4E-3	792.0E-3	1.476E+0	3.38E+0	6.5E+0	15.6E+0
Modeled Component Time													
Switch Delay	-	$R_0 * M$	s	G	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0	000.0E+0
MPI Contention	-	$(R_k - D_k) * M * P$	s	G	429.573E-15	10.003E-3	59.955E-3	233.847E-3	577.15E-3	1.444E+0	3.121E+0	7.155E+0	14.277E+0	32.99E+0
MPI Active	-	$D_k * M * P$	s	G	55.0E-6	75.494E-3	147.607E-3	374.187E-3	784.697E-3	1.854E+0	3.848E+0	8.75E+0	17.29E+0	40.115E+0
Compute Time	-	MsgCompute * M	s	G	216.745E-3	209.38E-3	157.0E-3	228.4E-3	362.4E-3	792.0E-3	1.476E+0	3.38E+0	6.5E+0	15.6E+0

Types	Description
A	Auxillary calculation, done separately
B	Values for building a model
C	Calculated in this spreadsheet
G	Ancillary calculation for grapic
I	Input directly from measurement data
R	Results from model
O	Output for building a model
V	Value for validating a model

Appendix C – Sample Component Time Bar Charts

Below, we show alternate views of the component times shown in Chapter V, Section B for MCR. These views show the aggregate values of the components as a single bar on the charts, with the outer bar representing the measured values from the target machine, and the inner bar representing the modeled values from QNM. We have also generated similar graphics for ALC and the Keck Cluster, as well as graphics plotting the behavior of the various classes given a specific processor allocation.

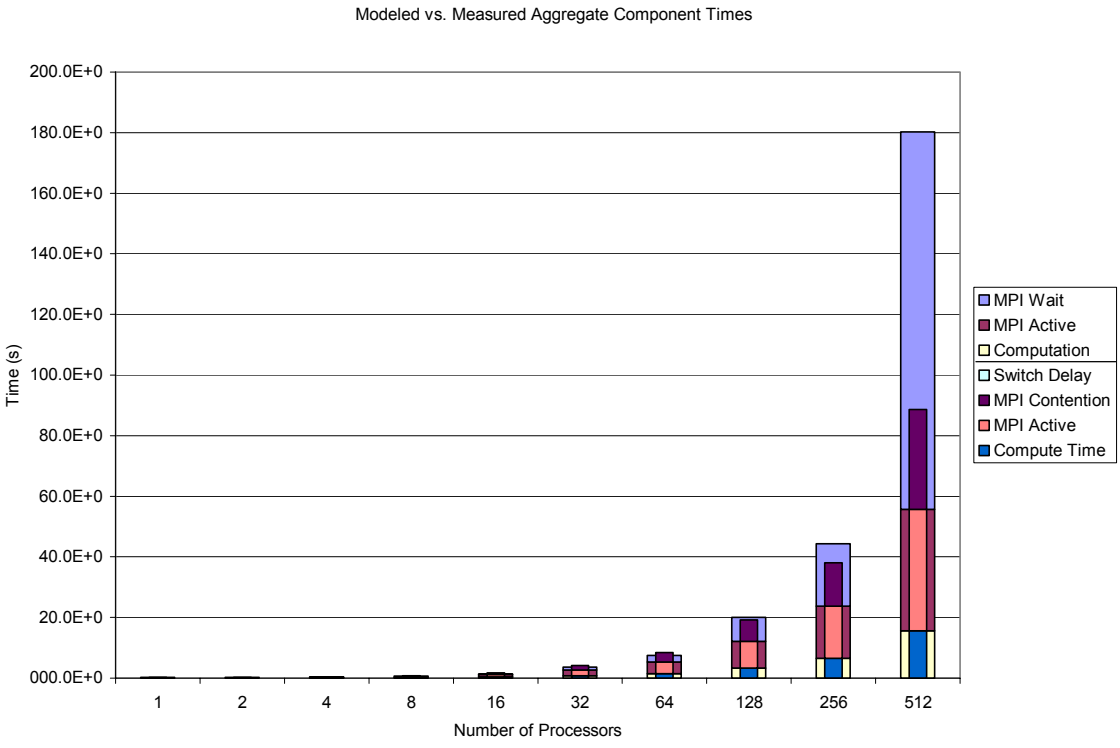

Figure 55 – MCR Class S Component Times

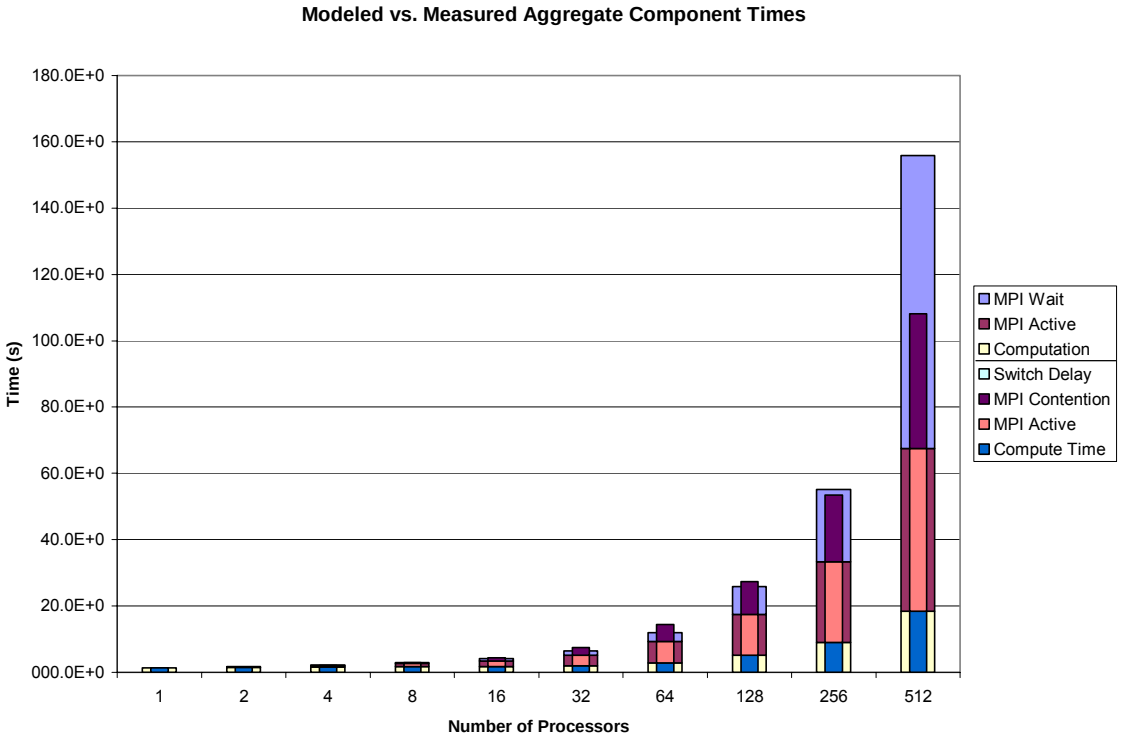

Figure 56 – MCR Class W Component Times

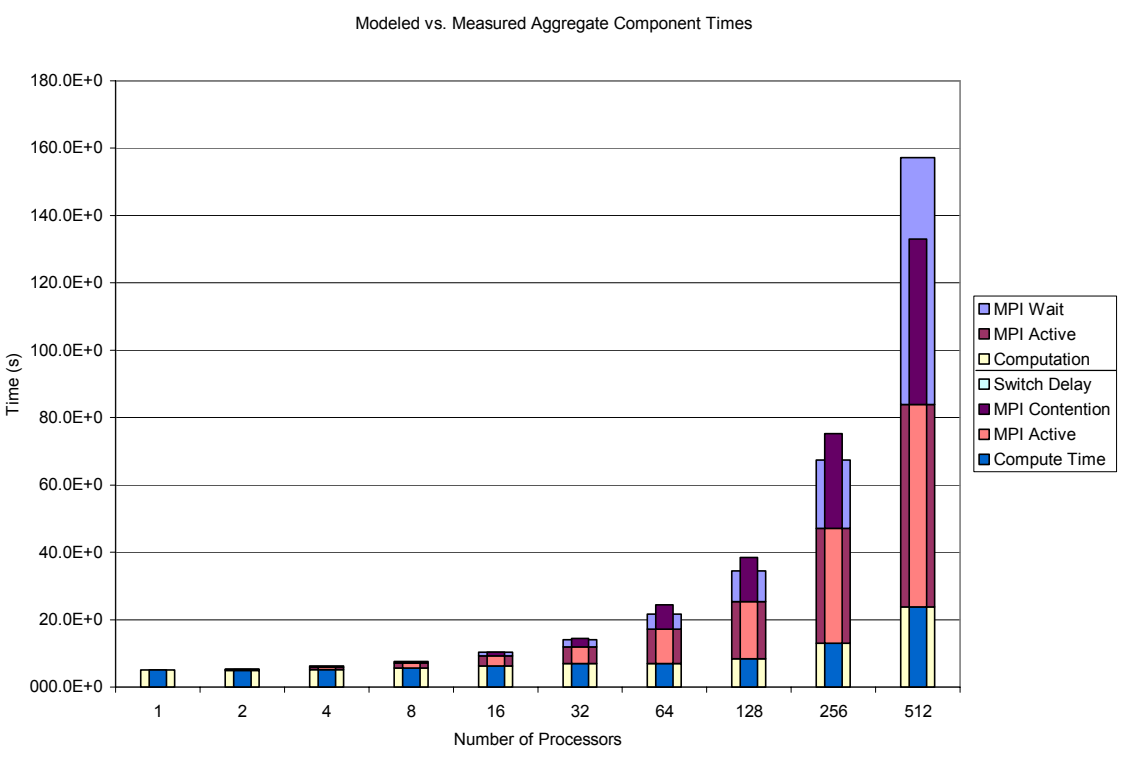

Figure 57 – MCR Class A Component Times

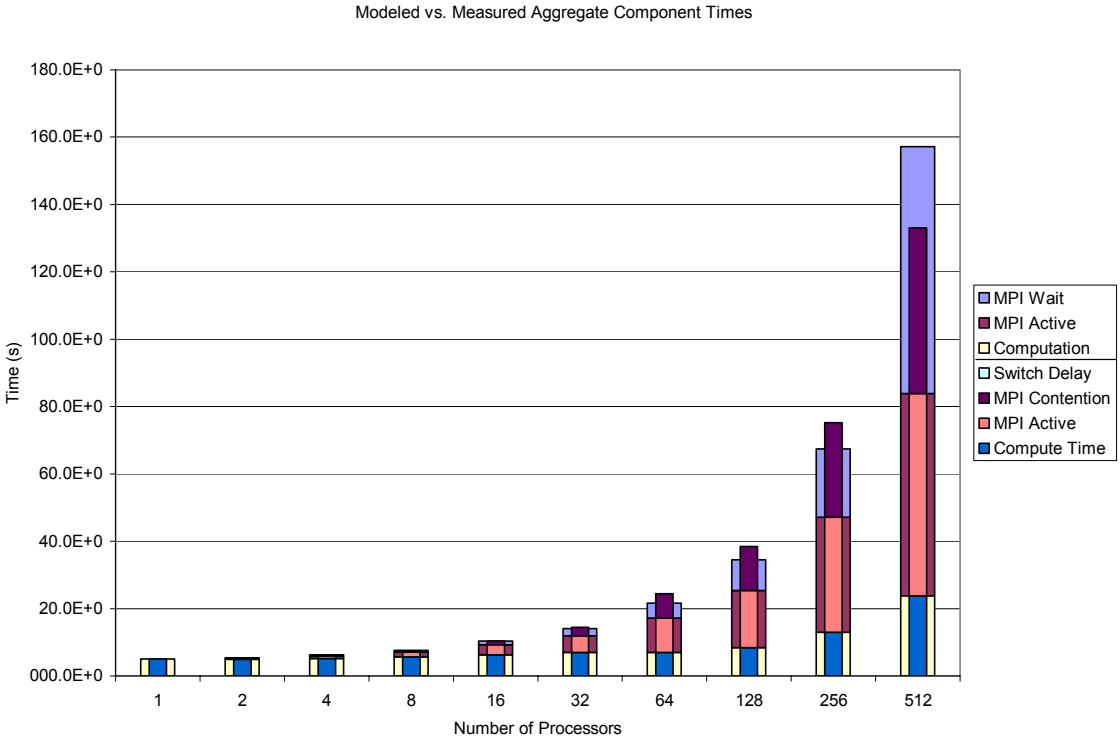

Figure 58 – MCR Class B Component Times

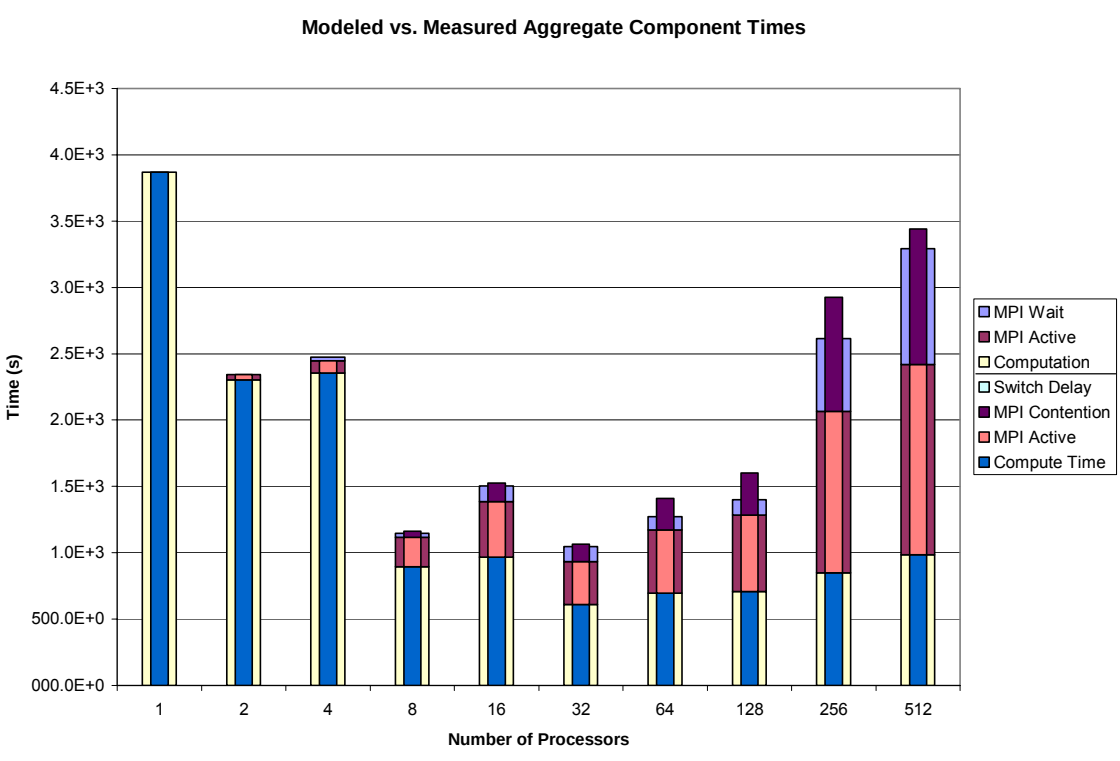

Figure 59 – MCR Class C Component Times

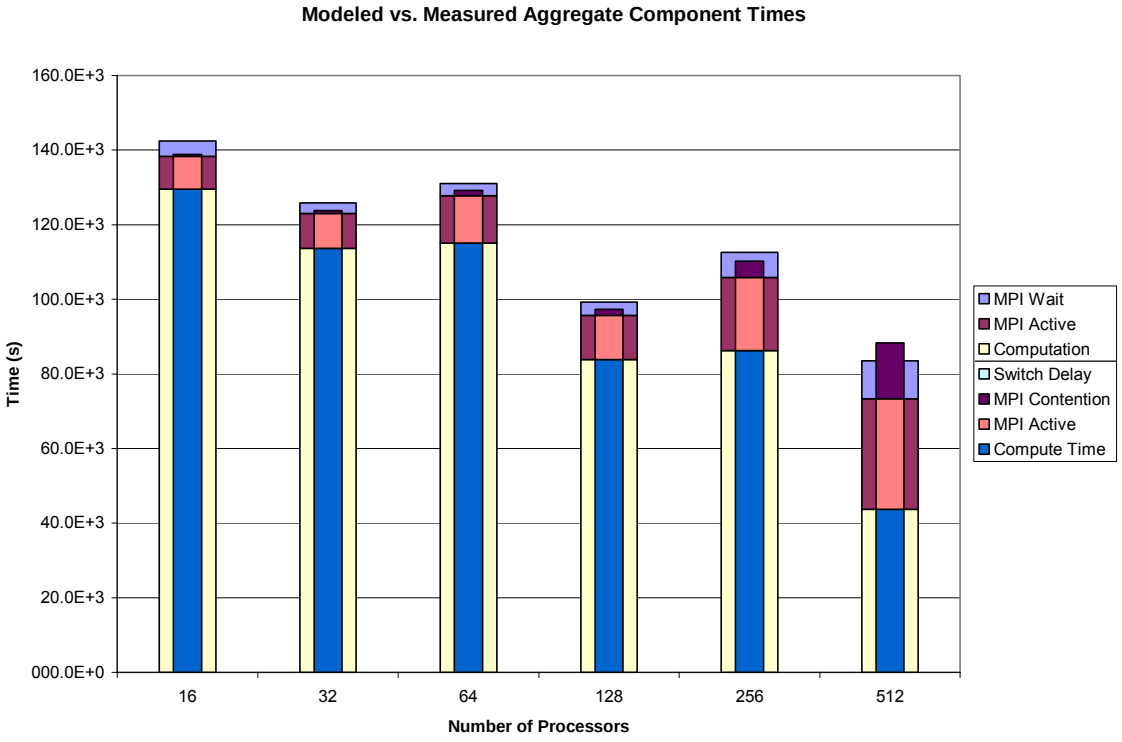

Figure 60 – MCR Class D Component Times

VIII – Bibliography

[ASC, 2004] ASC Linux Cluster (ALC). 25 Oct. 2004. Lawrence Livermore National Laboratory. 5 Feb. 2006. <http://www.llnl.gov/linux/alc/>.

[Alexandrov, 1995] Alexandrov, Albert, et al. "LogGP: Incorporating Long Messages into the LogP Model – One step closer towards a realistic model for parallel computation." ACM 7th Annual Symposium on Parallel Algorithms and Architectures. July 1995. Santa Barbara, CA: U. of California at Santa Barbara, 1995. <http://citeseer.ist.psu.edu/cache/papers/cs/2693/ftp:zSzzSzftp.cs.ucsb.eduzSzpubzSzpaperszSzschauserzSz95-spaa.pdf/alexandrov95loggp.pdf>.

[Bailey, 1994] Bailey, D., et al. "The NAS Parallel Benchmarks." RNR Technical Report. NASA Ames Research Center, Mar. 1994.

[Bailey, 1995] Bailey, D., et al. "The NAS Parallel Benchmarks." Report. NASA Ames Research Center, Dec. 1995.

[Balsa, 1997] Balsa, André. Linux Benchmarking – Concepts. "3. FPU tests: Whetstone and Sons, Ltd." 21 Sept. 1997. The Linux Gazette. 26 Nov. 2004. <http://www.tux.org/~balsa/linux/benchmarking/articles/html/Article1d-3.html>.

[Beowulf, 2005] Beowulf.org: Overview. "What Makes a Cluster a Beowulf?" 2005. Beowulf.org. 30 Mar. 2006. <http://www.beowulf.org/overview/index.html>.

[Bramer, 2004] Bramer, Brian. System Benchmarks. DeMontfort University, UK. 26 Nov. 2004 <http://www.cse.dmu.ac.uk/~bb/Teaching/ComputerSystems/SystemBenchmarks/BenchMarks.html>.

[Culler, 1999] Culler, David E, Jaswinder Pal Singh, and Anoop Gupta. Parallel Computer Architecture: A Hardware/Software Approach. San Francisco: Morgan Kaufmann Publishers, 1999.

[Culler, 1993] Culler, David, et al. "LogP: Towards a Realistic Model of Parallel Computation." ACM 4th SIGPLAN symposium on Principles and Practice of Parallel Computing. July 1993. Berkeley, CA: U of California at Berkeley, 1993.

<http://citeseer.ist.psu.edu/cache/papers/cs/979/http:zSzzSznow.cs.berkeley.eduzSz~demmelzSzcs267zSzlogp.pdf/culler93logp.pdf>.

[Dongara, 2004a] Dongara, Jack. Netlib Repository at UTK and ORNL. "Frequently Asked Questions on the Linpack Benchmark and Top500." 28 Oct. 2004. AT&T Bell Laboratories, et al. 26 Nov. 2004 <http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html#_Toc27885709>.

[Dongara, 2004b] Dongara, Jack. Netlib Repository at UTK and ORNL. "Linpack." 28 Oct. 2004. AT&T Bell Laboratories, et al. 26 Nov. 2004 <http://www.netlib.org/linpack/>.

[Faulkner, 2005] Faulkner, Sheila A. "Simple MPI Performance Measurements III." Paper. Lawrence Livermore National Laboratory, 2005.

[Frank, 1997] Frank, Matthew I., Anant Agarwal, and Mary K. Vernon. "LoPC: Modeling Contention in Parallel Algorithms." ACM 6th SIGPLAN Symposium on Principles and Practice of Parallel Computing. June 1997. Las Vegas, NV, 1997. <http://citeseer.ist.psu.edu/cache/papers/cs/3304/ftp:zSzzSzftp.cag.lcs.mit.eduzSzmfrankzSzppopp97.pdf/frank97lopc.pdf>.

[Grama, 2003] Grama, Ananth, et al. Introduction to Parallel Computing. 2nd ed. Harlow, England: Addison-Wesley, 2003.

[Ipek, 2005] Ipek, Engin, et al. An Approach to Performance Prediction for Parallel Applications. 16 Jan. 2007. Euro-Par 2005. 30 Aug. – 2 Sep. 2005. Lisbon, Portugal. <http://www.springerlink.com/content/ay63wtdah19m036g/?p=6a2ac886097a4713934b378d29a02e93&pi=23>.

[Ipek, 2006] Ipek, Engin, et al. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. ASPLOS 06. 21 – 25 Oct. 2006. San Jose, CA.

[Kanellos, 2004] Kanellos, Michael. "Intel kills plans for 4GHz Pentium." Tech News on ZDNet. 14 Oct. 2004. 18 Dec. 2006. <http://news.zdnet.com/2100-9584_22-5409816.html>.

[Keck, 2004a] Keck Cluster [USF-CS]. 27 Jul 2004. U. of San Francisco. 8 Aug. 2005. <http://kc.cs.usfca.edu/index.shtml>.

[Keck, 2004b] Keck Cluster [USF-CS]. 25 Jun 2004. U. of San Francisco. 8 Aug. 2005. <http://kc.cs.usfca.edu/software.shtml>.

[Lazowska, 1984] Lazowska, Edward D., et al. Quantitative System Performance: Computer System Analysis Using Queueing Network Models. Englewood Cliffs, NJ: Prentice-Hall, 1984. <http://www.cs.washington.edu/homes/lazowska/qsp/>.

[Lee, 2006] Lee, Benjamin, and David Brooks. Accurate and Efficient Regression Modeling For Microarchitectural Performance and Power Prediction. ASPLOS 06. 21 – 25 Oct. 2006. San Jose, CA.

[Linux, 2006] Linux Basics. 9 Feb. 2006. Lawrence Livermore National Laboratory. 23 Mar. 2006. <http://www.llnl.gov/linux/linux_basics.html>.

[LCOCF, 2006] LC Open Computing Facility – OCF. 28 Feb. 2006. Lawrence Livermore National Laboratory. 23 Mar. 2006. <http://www.llnl.gov/computing/hpc/resources/OCF_resources.html >.

[M&IC, 2002] M&IC Capability Cluster (MCR). 29 Aug. 2002. Lawrence Livermore National Laboratory. 12 Aug. 2005. <http://www.llnl.gov/linux/mcr/background/mcr_background.html#sec212>.

[M&IC, 2004] M&IC Capability Cluster (MCR). 6 Aug 2004. Lawrence Livermore National Laboratory. 8 Aug. 2005. <http://www.llnl.gov/linux/mcr/mcr.html>.

[Mauer, 2004] Mauer, Hans, et al. Top 500 Supercomputer Sites. 2004. Top 500 Supercomputer Sites. 26 Nov. 2004 <http://www.top500.org/lists/linpack.php>.

[MpiP, 2005] mpiP: Lightweight, Scalable MPI Profiling. 29 Apr. 2005. Lawrence Livermore National Laboratory. 8 Aug. 2005. <http://www.llnl.gov/CASC/mpip/>.

[NAS-PB, 2004] The NAS Parallel Benchmarks. 13 Oct. 2004. National Aeronautics and Space Administration Advanced Supercomputing Division. 8 Aug. 2005. <http://www.nas.nasa.gov/Software/NPB/>.

[Purple, 2001] The ASCI Purple Benchmarks. 26 Nov. 2001. Lawrence Livermore National Laboratory. 13 Nov. 2006. <http://www.llnl.gov/asci/purple/benchmarks/>.

[Snavely, 2001] Snavely, Allen, Laura Carrington, and Nicole Wolter. Modeling Application Performance by Convolving Machine Signatures with Application Profiles. IEEE Workshop on Workload Characterization. Dec. 2001. Austin, TX. 16 Jan. 2007. <http://www.sdsc.edu/~allans/micro.pdf>.

[SPEC, 2006] SPEC CPU2000. 14 Nov. 2006. Standard Performance Evaluation Corporation. 24 Nov. 2006. <http://www.spec.org/cpu2000/>.

[Tvrdik, 1999] Tvrdik, Pavel. CS838: Topics in Parallel Computing. "Section \#2: PRAM Models." 23 Jan. 1999. University of Wisconsin – Madison. 10 Dec. 2004. <http://www.cs.wisc.edu/~tvrdik/2/html/Section2.html>.

[Weboped, 2004] Dhrystone. "What is Dhrystone? – A Word Definition From the Webopedia Computer Dictionary." Webopedia.com. 26 Nov. 2004. <http://www.webopedia.com/TERM/D/Dhrystone.html>.

Index

N

O

P

Q

R

S

T

W